UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

A Systematic Analysis and Hardening of the Java Security Architecture

PhD Thesis
to obtain the degree of
"Doktor der Ingenieurwissenschaften (Dr.-Ing.)"

by
Philipp Albert Holzinger
born in
Neustadt an der Weinstrasse

Advisor:
Prof. Dr. Eric Bodden

Paderborn, September 10, 2019

Abstract

Java is one of the most popular development platforms and it is applied in a broad range of different application contexts. The Java Runtime Environment (JRE) implements a complex security architecture that enforces security policies in such a way that untrusted code can run along trusted code within the same process. However, over the course of its entire lifespan, a large number of attacks revealed many severe security vulnerabilities in the JRE that allowed for a full bypass of all security mechanisms.

Despite the many examples of security vulnerabilities in the platform, only little was previously known about conceptual commonalities of different exploits and the extent to which design weaknesses in the Java security architecture enabled the attacks. Thus, in this work, we systematically collected and analyzed a large body of exploits for different versions of the JRE, covering vulnerabilities of more then ten years. One result of this analysis is that there is a set of nine commonly abused weaknesses, and we further show that all exploits in the sample set can be divided into three categories of attacks. Finally, we identified two major design weaknesses that enabled many of the attacks.

The first design flaw is weak information hiding. We found that the security of the entire Java platform rests on the confidentiality and integrity of individual variables in system classes. At the same time the security architecture lacks defense in depth, which allows that individual implementation defects can break information hiding, thus undermining all security guarantees. To address this problem, we proposed a lightweight mitigation strategy that our proof-of-concept implementation can integrate into even closed-source JREs. Our evaluation showed that this solution systematically blocks 84% of the information-hiding attacks contained in our exploit sample set, and we also explained how the remaining attacks can be blocked as well. We further showed that our solution is backward compatible and the runtime overhead induced by integrating the countermeasures is low. In addition to this lightweight mitigation strategy, we further presented a heavyweight mitigation strategy. This alternative solution suggests a comprehensive redesign of the internals of the Java runtime. As we explain, implementing this mitigation strategy would require access to the JRE's source code and major engineering efforts. However, this heavyweight solution has the potential to fundamentally strengthen information hiding in the Java platform, and outperform our lightweight proof of concept in terms of both robustness and speed.

The second design flaw we identified besides weak information hiding is improper access control, which is manifested in various different ways. In particular, we found that several

sensitive methods in Java system classes implement what we call "shortcuts"—they skip proper permission checks if certain hardcoded constraints on the call stack are satisfied. As we show, this approach to implementing access control is error-prone, increases the attack surface, and decreases code maintainability. To address this problem, we created a variant of the Java runtime that works almost without shortcuts, whereby privileged blocks become the standard way for elevating privileges. As we explain, this substantially facilitates code maintenance, as well as automatic and manual program analysis. Also, certain attack vectors are blocked by this solution. Through a large-scale set of experiments we show that our proposed changes have virtually no impact on the performance of a set of real-world applications. We finally assessed the impact of moving to a shortcut-free platform for productive use by discussing usability and backward compatibility considerations, and also presented lessons learned that may serve as a guidance for the design and implementation of other complex security architectures.

Zusammenfassung

Java ist eine der beliebtesten Entwicklungsplattformen und wird in einem breiten Spektrum unterschiedlicher Anwendungskontexte eingesetzt. Das Java Runtime Environment (JRE) implementiert eine komplexe Sicherheitsarchitektur, die die Einhaltung von Sicherheitsrichtlinien derart sicherstellt, dass die Ausführung von vertrauenswürdigem Code und nicht vertrauenswürdigem Code im selben Prozess ermöglicht wird. Über die gesamte Lebenszeit der Plattform gab es jedoch wiederkehrend Angriffe, die schwerwiegende Sicherheitslücken in der JRE aufzeigten, mit deren Hilfe alle Sicherheitsmechanismen umgangen werden konnten.

Trotz der großen Anzahl bekannter Sicherheitslücken in der Plattform war bisher wenig bekannt über konzeptionelle Gemeinsamkeiten unterschiedlicher Exploits, oder in welchem Umfang Designschwächen in der Java-Sicherheitsarchitektur die Angriffe erst ermöglichten. Daher haben wir im Kontext dieser Arbeit eine systematische Sammlung und Analyse eines großen Datensatzes unterschiedlicher Exploits durchgeführt, wodurch wir Sicherheitslücken aus mehr als zehn Jahren betrachteten. Ein Ergebnis dieser Analyse ist, dass es neun Schwächen gibt, die häufig von Exploits ausgenutzt werden. Zudem zeigen wir, dass alle Exploits in drei Angriffskategorien eingeteilt werden können. Abschließend verweisen wir auf zwei fundamentale Designschwächen, die zu einer Vielzahl von Angriffen führten.

Die erste dieser Designschwächen ist die spezifische Gestaltung und Implementierung von Information Hiding in der Laufzeitumgebung. Wir konnten zeigen, dass die Sicherheit der gesamten Plattform auf der Vertraulichkeit und Integrität einzelner Variablen von Systemklassen ruht. Gleichzeitig fehlt der Sicherheitsarchitektur jedoch ein mehrschichtiges Sicherheitskonzept, was zur Folge hat, dass einzelne Implementierungsfehler den Zugriff auf sensitive Variablen erlauben, und somit alle Sicherheitsgarantien untergraben werden können. Um dieses Problem zu adressieren haben wir einen leichtgewichtigen Verteidigungsansatz entwickelt, der selbst in JREs integriert werden kann, für die der Quelltext nicht vorliegt. Unsere Evaluation zeigte, dass dieser Ansatz systematisch 84% der Information-Hiding-Angriffe in unserem Datensatz blockieren konnte, und wir erklären wie die verbleibenden Angriffe ebenfalls adressiert werden können. Zudem zeigen wir, dass unsere Lösung rückwärtskompatibel ist und nur einen geringen Einfluss auf die Ausführungsgeschwindigkeit hat. Zusätzlich zu diesem leichtgewichtigen Lösungsansatz präsentieren wir einen alternativen schwergewichtigen Ansatz, dessen Implementierung eine umfassende Neustrukturierung der JRE bedarf. Die Umsetzung dieser Änderung würde den Zugriff auf den Quelltext der JRE erfordern und würde mit einem erheblichen Implementierungsaufwand einhergehen. Wie wir erklären hat dieser schwergewichtige Ansatz jedoch

das Potential die Implementierung von Information Hiding in Java erheblich zu stärken, und unseren leichtgewichtigen Ansatz mit Hinblick auf Robustheit und Geschwindigkeit zu übertreffen.

Die zweite Designschwäche, die wir im Rahmen unserer Exploitanalyse identifiziert haben ist inkonsequente Zugriffskontrolle, die sich auf verschiedene Weise zeigt. Im Besonderen konnten wir zeigen, dass einige sensitive Methoden in Java-Systemklassen sogenannte "Abkürzungen" implementieren—sie überspringen eine ordnungsmäßige Zugriffsprüfung, wenn bestimmte fest programmierte Bedingungen bezüglich des Aufrufstapels erfüllt sind. Wir konnten zeigen, dass diese Art der Zugriffskontrolle fehleranfällig ist, die Angriffsfläche vergrößert, sowie die Wartbarkeit der Codebasis verringert. Um diesem Problem zu entgegnen haben wir eine Variante der JRE erstellt, die weitgehend ohne "Abkürzungen" funktioniert und stattdessen die Privileged-Block-API für die Erhöhung von Privilegien verwendet. Wir erklärten, dass dadurch die Wartbarkeit des Quelltextes, sowie die automatische und manuelle Programmanalyse vereinfacht werden. Zudem wurden bestimmte Angriffsvektoren durch die Änderung blockiert. Durch umfangreiche Experimente konnten wir zeigen, dass die vorgeschlagenen Änderungen die Ausführungsgeschwindigkeit einer Auswahl komplexer Anwendungssoftware nur unwesentlich beeinflusst haben. Zusätzlich haben wir die Auswirkungen diskutiert, die ein Verzicht auf "Abkürzungen" im Produktiveinsatz mit sich bringen würde. Dies umfasst Betrachtungen mit Hinblick auf Usability und Rückwärtskompatibilität. Abschließend haben wir die gewonnenen Erkenntnisse im breiteren Kontext betrachtet und geben damit Hinweise auf die sichere Gestaltung und Implementierung anderer komplexer Sicherheitsarchitekturen.

Publications

This dissertation is an original work. Parts of it, however, have already been published directly or in similar form in a set of research papers, for which the author of this thesis is also the lead author. Specifically, this includes the following works:

- P. Holzinger, S. Triller, A. Bartel, and E. Bodden. An in-depth study of more than ten years of Java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 779–790. ACM, 2016
  Several parts of Chapter 3 were taken directly or with modifications from this paper.

- P. Holzinger and E. Bodden. A Systematic Hardening of Java's Information Hiding. To be published
  Several parts of Chapter 4, as well as Section 6.2.2 were taken directly or with modifications from this paper.

- P. Holzinger, B. Hermann, J. Lerch, E. Bodden, and M. Mezini. Hardening Java's Access Control by Abolishing Implicit Privilege Elevation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1027–1040, May 2017
  Large parts of Chapter 5 were taken directly or with modifications from this paper.

Acknowledgments

First and foremost, I want to thank my advisor, Eric Bodden. His continued support has been a great help in advancing my research. At all times his guidance and contributions have been of major value. Particularly noteworthy is his excellent scientific expertise, but also his individual personality, which has always contributed to a creative and productive working environment. Also, I want to thank my coauthors Alexandra Bartel, Ben Hermann, Johannes Lerch, Mira Mezini, and Stefan Triller for all the constructive discussions we had. Further, I want to thank Fraunhofer SIT and the past and present members of the SSE group who provided me with a supportive environment.

Part of this work was supported by an Oracle Research Grant. In particular, I would like to thank Cristina Cifuentes and Andrew Gross from Oracle for valuable insights into various aspects of the Java security architecture, and their feedback on our work. Also, I want to thank Julian Dolby from IBM for providing us with installation packages for IBM's JDK, which we used for analyzing the behavior of exploits.

Finally, I want to thank my mother for always believing in me, and the many friends with whom I've spent great times away from research.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction and contributions

As an introduction to the topic of this thesis, this chapter provides a brief overview of the history of the Java technology and explains its impact on software engineering as a discipline. We further outline our research motivation and present a thesis statement. Moreover, we specify the scope of our work, present challenges, review related work, and conclude with an overview of the main contributions of this thesis.

1.1 Introduction

The Java technology has seen widespread adoption in the past decades and it significantly influenced software development as a discipline. In 2015, the Institute of Electrical and Electronics Engineers (IEEE) awarded James A. Gosling, the principal inventor of Java, the IEEE John von Neumann Medal, stating that his "development of the Java programming language in 1995 was a major milestone in computing that has had an immeasurable impact on computer science. Dr. Gosling combined the best ideas in programming languages with his own ideas to create the first widely deployed programming language featuring portability to allow transmission of code over the Internet from one computer to another for execution while still meeting security requirements." [54]

Java is a collective term that refers to a programming language, a development kit (Java Development Kit, JDK), and a runtime environment (Java Runtime Environment, JRE). The foundation of this technology is a set of specifications [80], including "The Java Language Specification", and "The Java Virtual Machine Specification". It was originally developed at Sun Microsystems and presented to a larger audience in 1995 as part of the HotJava browser [27], the first browser to support Java applets. Java applets are small computer programs that web developers can integrate into their web pages to implement functionality and visualizations that, historically, web technologies like HTML and JavaScript were not able to deliver [28]. Using hardware acceleration and just-in-time compilation, Java applets were used to implement graphically demanding games and applications, but also utility software, like web-based remote shells, which enriched the browsing experience by features that browsers at the time were not able to provide. Netscape Navigator, one of the leading browsers at the time, provided support for Java applets shortly after the technology's release [27], which contributed to Java's popularization early on. Eventually, all major browser platforms supported the Java browser plugin that added support for applets, and the Java technology became ubiquitous on desktop workstations.

The more recent evolvement of new web and browser technologies has drastically reduced the need for Java applets and other plugin-based extensions, like Flash. As a consequence, in 2016, Oracle, who acquired the Java technology from Sun Microsystems, announced that the Java browser plugin would be deprecated and eventually removed from the Java runtime [83]. However, since Java was first presented to the public, its role has changed and it turned into a general development platform for a broad spectrum of applications, both on the client and server side. The Java Standard Edition executes standalone Java applications on desktop workstations and has seen widespread adoption on different operating systems, including Windows and Linux. The Java Enterprise Edition targets server systems and is used to develop and execute complex business applications. Besides, there is a set of Java editions that target other platforms, such as smartcards, and other embedded devices. According the TIOBE index [105], Java is the most popular development language as of August 2018, thus outranking C, C++, Python, and PHP, for example.

Software security was a major concern of Java's original design. Considering that the Java platform was designed to retrieve and execute code from potentially untrusted remote sources, such as applets embedded in websites, it was pivotal to restrict the behavior of untrusted Java code to actions that could not harm the host system that executed the Java runtime. Because of this, already the JDK 1.0 release implemented the so-called "sandbox model", which contained remote code in a restricted environment that did not allow for potentially harmful actions. Its name might be misleading, though, as the sandbox is actually a combination of various different mechanisms that need to work together to provide security guarantees. If, for example, the user of a browser loaded a website that embedded an applet, the mechanisms that made up the sandbox actively prevented the applet from accessing sensitive resources of the host system, such as the local file system. This original security model already featured several of the mechanisms that still operate in modern versions of Java, including, for example, the bytecode verifier, which sanity-checks all Java classes prior execution, or automatic memory management, which transparently allocates and deallocates memory to prevent low-level vulnerabilities in Java code.

However, compared to current Java editions, this early security model was very simplistic—it categorized code as being either local code, which had full access to the host system's resources, or remote code, which was executed only with very limited capabilities. Customizing the sandbox required in-depth knowledge of the platform and significant engineering effort. The release of JDK 1.1 introduced signed applets, which were considered trusted and treated like local code, which opened the platform to a set of use cases that the original sandbox model could not handle. A more substantial change to the Java security model, however, was introduced with the release of JDK 1.2. This version introduced a security model that is far more elaborate than the original sandbox model, and it is still in use in the modern versions of Java that we considered in this work. Its central feature is fine-grained access control that is used to enforce security policies that can be fully customized by users and administrators of the Java runtime without requiring implementation work. In contrast to earlier versions of Java, all code is equally subject to policy enforcement, whether it is local or remote code. This new security model is much more flexible than its predecessors, but also more complex.

Due to Java's broad distribution on desktop computers and servers, it soon became a major target for attackers. Applets embedded in websites that were able to exploit vulnerabilities

in the security mechanisms of the Java platform gained full access to the user's host system that executed the browser, which effectively compromised the entire system. A single malicious applet would have the potential to compromise an enormous number of users if it were embedded into a popular website. It is thus not surprising that Cisco's annual security reports show that Java was the primary attack vector for online criminals in 2013 [20] and 2014 [21]. Every single major release of the Java runtime had multiple severe security defects that allowed for a full bypass of all security mechanisms. This problem, however, goes beyond browser security alone. As empirical studies show [22], the security model is applied in a broad range of different applications, including development tools and business applications, such as Eclipse, Hadoop, Lucene, Spring, Struts, Tomcat, IntelliJ IDEA, and many more. Security defects in the Java runtime are thus a major concern of many companies and end users alike.

This thesis is concerned with a detailed analysis of Java's design weaknesses, and an exploration of different ways on how the platform can be systematically hardened to withstand common attacks. As we show, a fundamental problem with the security architecture is the fact that the different security mechanisms it contains do not sufficiently backup each other. In consequence of this design flaw, individual implementation defects can break all security guarantees that Java's security architecture seeks to provide. The knowledge gained through our study is on the one hand of practical use, as the Java platform is widely used and security issues potentially affect a large number of users and companies. Interestingly, our findings indicate that even for Java vendors it is generally hard to properly implement, maintain, or even use Java's security model, partially due to its conceptual complexity, but also due to certain design decisions. On the other hand, as the following section explains in more detail, the insights we gain are of a more foundational interest—this thesis can be seen as case study that extensively shows how a complex and long-living system like Java fails to solve a complex problem in the field of software security: securely containing untrusted code within the boundaries of a trusted environment. In summary, we hope that our study contributes to a better understanding of secure software design, thus enabling researchers and practitioners to develop more secure and reliable systems in the future, without having to repeat past mistakes.

1.2 Research motivation

The Java security model's overall goal is to contain the execution of untrusted code within a virtual environment such that the host system's processes and resources are protected from unauthorized access. In more general terms, the underlying fundamental concern is to implement privacy and security by means of logical *isolation*. Using isolation to implement security and enhance reliability is a well-known concept in the field of applied computer security that dates back at least to the era of early time-sharing systems. In 1965, Corbató and Vyssotsky declared the "protection of private files and isolation of independent processes" to be of "critical importance" to the design of Multics [23], a time-sharing operating system that heavily influenced the design of modern operating systems.

Nowadays, containing untrusted code by virtually isolating it from critical system resources is an important concern of many different categories of software. The following provides a set

of examples, and moreover, shows that in practice all those systems eventually fail, although the underlying problem has been known for decades.

- Malware analysis tools execute applications in a simulated environment that is isolated from other applications and the host operating system in order to dynamically analyze their behavior with the goal of determining whether the application is benign or malicious. It is pivotal for such a malware analysis tool to properly restrict the ways how the application under investigation can interact with the host system and its resources in order to prevent harm while conducting the analysis. One prominent example of a tool that provides such capabilities is Windows Defender which is shipped with consumer editions of Microsoft Windows. It contains various interpreters and emulators that are used to dynamically analyze potentially malicious applications. In 2017, a team of Google researchers found a severe vulnerability in its JavaScript interpreter that allowed for privilege escalation on a target machine without requiring interaction from the user [73]. The root cause of the problem was a type confusion vulnerability [85] filed under CVE-2017-0290.
- One of the most popular file formats for exchanging documents is the Portable Document Format (PDF). The PDF standard provides for the integration of JavaScript code in documents [6] to allow for the implementation of interactive elements in documents that dynamically respond to user interactions. Since PDF viewers are also commonly used to display files originally received from untrusted sources, it is of high importance to properly contain any code segments included in documents. In 2015, a vulnerability was discovered in Adobe Acrobat Reader that allowed malicious JavaScript code in documents to escalate privileges [77].
- While browsing the web, modern browsers download untrusted code from web servers and execute it on the end user's machine. To ensure that malicious code embedded in websites cannot harm the host system, browsers attempt to contain the code in an isolated environment with a reduced set of privileges. However, all of the most popular browsers had severe security vulnerabilities in the past that allowed untrusted code to perform actions that would otherwise be prevented by the browsers' security policies. One prominent example for the Firefox browser is the combination of CVE-2014-1510 and CVE-2014-1511, which allowed an attacker to escalate privileges of JavaScript code, thus allowing for arbitrary code execution [76]. Another more recent example is a type confusion vulnerability in the Chromium browser's webassembly engine [92], which was filed under CVE-2017-15413.
- Platform virtualization software is used to execute several virtual machines (VMs) in parallel on top of shared hardware. It is a desired feature that VMs are logically isolated from each other and the host system that executes the virtualization software, as VMs and the code they execute may be untrusted. Although a secure containment of VMs is a fundamental feature of virtualization software, many well-known products frequently contain so called "VM escape vulnerabilities" that allow code executed within VMs to escalate privileges and compromise the host system or other VMs. For example, in 2018, several such vulnerabilities were found in Oracle Virtual Box [84]. In 2017, several vulnerabilities with a similar impact were found in Xen, e.g., CVE-2017-8904 / XSA-214.
- Finally, Android, a widely distributed mobile operating system, is partially implemented in Java, and so are many third-party apps. Due to the fact that apps may be provided by

untrusted parties, and users may trust certain apps more than other apps, it is important to properly restrict the functionalities that apps can use, reliably isolate app code from system code, and also isolate apps from one another. In contrast to the standard editions of Java that we study in this work, Android uses another security model that leverages kernel-level access control by means of process isolation. However, in the past, also Android was prone to, e.g., critical remote code execution vulnerabilities that allowed attackers to execute code in privileged contexts [43]. Section 6.2 provides more details on Android's approach to access control and the extent to which our findings can be transferred also to this platform.

The above list of software categories and vulnerabilities is far from exhaustive, but it shows the relevance of secure code containment for a broad range of software products. It further illustrates that even professionally-developed and widely-used software solutions were frequently prone to severe security vulnerabilities that allowed malicious code to escape their restricted containers which implies significant room for improvement. An investigation of why secure code containment fails in practice is certainly required to shed light on this up-to-date problem. Further, any insights into how systems can be hardened to withstand common attacks are of practical and academic importance.

A major concern of this thesis is to contribute to a better understanding of how secure runtime environments can be designed and implemented such that they reliably deliver fundamental security guarantees. We decided to take a depth-first approach that specifically focuses on one selected runtime as opposed to a breadth-first approach that focuses on a comparative analysis of various different runtimes. Although both perspectives are of academic interest and have the potential to advance the state of the art, we must make a decision for one of them because our overall resources are limited and thorough research into either direction is labor-intensive.

We chose Java as a valuable target for our explorative research for a number reasons. This specifically includes:

- Widely distributed—According to Oracle, Java is running on 15 billion devices [79]. It is hard to independently evaluate this number and break it down by different categories of devices, but it is certainly reasonable to assume that even just the desktop and server editions of the JRE are installed on an enormously large number of computers.
- Highly relevant—Cisco reported that Java was the top attack vector used by online criminals in 2013 [20] and 2014 [21]. Especially security vulnerabilities in the desktop edition of the JRE receive high attention by the general public and authorities alike. For example, Germany's Federal Office for Information Security issued a warning concerning Java Applets and Java Web Start applications and recommends to disable Java in browsers [13]. Although this is an unfortunate fact, it still highlights Java's importance for a broader audience.

  Moreover, Java is used by millions of developers and students world-wide as reported by Oracle [79], and it is the most popular development language as of August 2018 according the TIOBE index [105]. A search on Scopus [2] for scientific publications with the term "Java" in the title and either "ACM" or "IEEE" in the conference name yields more than 2800 publications.[1]

[1] Search performed on 05-Feb-2019. Exact query string used: (TITLE(Java) AND CONF(ACM)OR CONF(IEEE))

- Long-living—We outlined in Section 1.1 that Java was already released in the mid-1990's. Therefore, there is a correspondingly long history of security incidents and exploits that can potentially be used for empirical studies. Also, since Java's popularity is high, there is a chance that the runtime will be further maintained for several years or even decades, and any insights we gain during our studies that contribute to its security may have a lasting impact.
- Powerful—The Java security model is feature-rich and enforces fine-grained security policies for applications written in a complex, general-purpose programming language. Therefore, there is a fair chance that the insights we gain for the security of Java may be generalizable and also relevant to other runtimes and systems.
- Complex—The Java language and runtime are sufficiently complex to assume that the problems that lead to the high number of security vulnerabilities are far from trivial.

1.3 Context and scope

This thesis focuses on an in-depth analysis and hardening of the Java runtime's security architecture. The foundation of this architecture is the Java security model that has been extensively documented, for example by Gong et al. [39]. The reference implementation for the security architecture is the OpenJDK [1], but we will also consider proprietary implementations, specifically those by Oracle and IBM, as they are widely distributed. Various editions of the Java runtime have been released in the past, but our focus will be on the "Java Platform, Standard Edition" (Java SE). It implements the full security model as described by Gong et al. [39] and it is capable of executing most Java applications. With respect to platform security, it is equivalent to the "Java Platform, Enterprise Edition" (Java EE) that extends Java SE by features that are relevant to enterprise server applications. Other editions of the platform, for example Java Card or Java ME, are functionally limited and therefore not in our scope. We further set a focus on the Java releases that were actively maintained at the time we conducted our study, thus including Java 6, 7, and 8.

1.4 Challenges and risks

The plan of analyzing and hardening the security architecture of the Java platform is subject to a number of challenges and involves a set of risks. Challenges that we want to highlight are:

- Size and complexity—The Java runtime comprises several thousand Java classes and a considerable amount of native code. This implies that implementing certain analyses and transformations of the runtime require automation, as the code base is too large for exhaustive manual reviews and edits. Due to the system's size and complexity, we expect that any change that we propose to improve the platform will require non-trivial changes to its implementation and a thorough analysis to rule out any unwanted side effects.
- Backward compatibility—As Section 1.1 explains, the Java runtime has a history of more than two decades. A large number of Java-based applications were developed within this time and it is reasonable to assume that a large number of legacy applications that

are no longer actively developed are still in use today. Therefore, retaining backward compatibility is of high importance and a crucial aspect of any proposed change to the platform.

- Performance requirements—The Java runtime is a software that is used for productive purposes by private end users and business users alike. Java is in competition with languages like C and C++ that allow for the implementation of high performance applications and hardware-specific optimizations. The introduction of Java's just-in-time compiler was a necessary step to achieve a comparable execution speed. Any proposed change to the platform that would significantly decrease performance would reduce Java's value in comparison to alternative languages and is hence to be avoided.
- Different implementation languages—As will be explained in more detail later, the platform comprises different components, and some of them are implemented in Java, while others are implemented in native code. This combination of different technologies make program analysis more difficult and increase the effort required for implementing automated code transformations.
- Non-modularity—The Java security model comprises various different security mechanisms, but their implementation in the Java runtime is not modularized, i.e., security-related code is scattered across large portions of the entire code base [106]. This complicates reasoning about the platform's security properties and makes it hard to design and implement fundamental security-related changes that strengthen the platform.

In addition to these challenges, we face a number of risks. First of all, it is possible that by empirical analysis we find out that the security vulnerabilities and exploits for the Java platform are vastly different from one another, and that there are hardly any commonalities that would allow for the design of a systematic hardening that would mitigate a large number of attacks. However, based on our knowledge of the platform and past vulnerabilities, we assume that there is a fair chance that a thorough empirical analysis will reveal at least some overlap among the different attack vectors that are relevant for the Java platform, thus yielding a potential for a systematic mitigation strategy.

Another risk is that the Java runtime's code base is of such high quality that code maintainability cannot be increased in a way that contributes to security. In that case, it may be considered unlikely that simple code refactorings introduce security vulnerabilities. Taking the many software projects into account that face significant shortcomings in that respect, we assume that there is hidden potential also in the code base of the Java runtime that allows for security improvements by addressing the issue of low maintainability.

Finally, we face the risk that Java already implements the strongest possible defense against common attacks and that there cannot be any efficacious strengthening that mitigates attacks while, at the same time, retaining backward compatibility and high performance. After all, the Java security model has been researched for many years and significant effort has been spent on improving its design and implementation. It is therefore not unlikely that the system is already highly optimized with respect to security under the constraints imposed by its environment and the technologies it uses. However, we expect that a thorough analysis of the threat landscape and relevant attack vectors has the potential to reveal previously unknown shortcomings that can be addressed by novel solutions.

1.5 Thesis statement

This dissertation confirms the thesis statement that

> *The Java runtime's security architecture can be hardened to withstand common attacks while retaining backward compatibility and high performance.*

More specifically, we show that

TS1. The Java security architecture has design weaknesses that are commonly exploited by attackers.

TS2. The Java runtime can be transformed, in a backward-compatible manner, to defend against information-hiding attacks.

TS3. The Java runtime's maintainability can be increased to lower the risk that refactorings introduce security vulnerabilities.

1.6 Summary of contributions

This thesis makes the following original contributions.

1.6.1 Systematic large-scale analysis of Java exploits

We collected and analyzed a large and diverse set of Java exploits. The primary goal was to determine how attackers exploit the Java platform and to identify commonalities and differences among the different attacks so that we can reason about root causes and weak spots in the Java security architecture.

For this, we harvested various online databases and exploits frameworks. This resulted in a collection of 87 exploit samples, including exploits for the Oracle JRE, the IBM JDK, and one sample specifically for Apple's JDK, covering a time span of over ten years. After the collection process, we evaluated all exploits to ensure that they implement reproducible attacks. For this, we integrated them into a custom-made exploit framework and removed any code segments that were unnecessary for the actual attack, like payloads, for example. Any exploit samples that we could not reproduce were removed from the collection. Then, we merged semantically equivalent samples that we originally received from different sources to ensure that the final sample set contained only unique and reproducible exploits. The result of this collection process was a total set of 61 exploits. To the best of our knowledge, this is the largest sample set of Java exploits to date.

We further developed a meta model to document exploit behavior on a conceptual level by means of several building blocks, so called primitives. This allowed us to document and compare different exploits on a level more abstract than the implementation. On the level of primitives, we can find commonalities among different exploits that, e.g., abuse different vulnerabilities, but

generally implement the same attack vector. As we explain, this meta model is not specific to Java, but sufficiently general to be applied in different domains.

Our analysis revealed that there is a small set of weaknesses in the Java security architecture that are abused by a large number of different exploits. We show that improper access control and weak information hiding are two major design weaknesses that enabled many of the attacks we observed. For example, several exploits abuse confused deputies and caller sensitivity to obtain unauthorized access to private methods and fields in system classes, which is sufficient to bypass all security mechanisms. This knowledge and other insights we gained enabled us to understand why the Java security model fails in practice, and thus serve as a foundation for the other contributions of this thesis.

1.6.2 Mitigating attacks on information hiding

The Java security architecture comprises a set of different security mechanisms. One may assume that these individual mechanisms are designed to backup each other, such that in case one mechanism fails due to an implementation defect, at least one other mechanism can compensate for the flaw. As Pompon [91] explains, one must "assume breach" when designing a secure software system, such that individual defects can only have a very limited impact on the security of an entire system. The concept of *defense in depth*, i.e., the implementation of multiple redundant countermeasures, has therefore gained widespread adoption in safety or security critical systems, inside and outside the software engineering discipline.

However, we show that, instead of implementing defense in depth, Java's security mechanisms provide largely orthogonal security guarantees. Consequently, a single vulnerable mechanism voids the security of the entire platform. We identified that Java's implementation of information hiding is especially brittle because it requires a set of different mechanisms to work together. At the same time, as we show, information hiding is fundamentally important for the security of the platform.

To study the extent to which information-hiding attacks can be mitigated, and at what cost, we give detailed account of two possibilities to strengthen information hiding such as to block all attacks known to date. The first solution strategy is heavyweight in the sense that is is clean and simple to explain, but requires deep and extensive modifications of the Java runtime's internals. Implementing this solution would require access to the target JRE's source code.

To be able to conduct large-scale experiments on various different Java runtimes, including closed-source JREs, we also present a lightweight solution strategy. In contrast to the heavyweight strategy, the lightweight solution can be automatically integrated into even closed-source Java platforms. We implemented a proof of concept that integrates a set of countermeasures into a given Java runtime without requiring access to its source code. Our evaluation shows that this hardening blocks 84% of all information-hiding attacks in our sample set, and we explain how also the remaining exploits can be blocked as well. We further show that our changes retain backward compatibility and high performance—a sample set of complex real-world applications runs unaffectedly with an average performance overhead below 2%. Finally, we explain that, if implemented, the heavyweight solution strategy has the potential to improve over the lightweight strategy in terms of both speed and robustness.

1.6.3 Hardening access control by abolishing implicit privilege elevation

We show that the Java security architecture implements shortcuts that bypass Java's general principle of stack-based access control under certain circumstances. Presumably, these shortcuts were originally introduced to improve performance. However, as we find, these shortcuts represent an implicit way of privilege elevation that decreases code maintainability of the platform to a point that simple code refactorings are likely to introduce severe security vulnerabilities. We conducted a tool-assisted adaptation of the platform that removes these shortcuts. By this, we move to a fully explicit model of privilege elevation which effectively hinders the introduction of new vulnerabilities and also restricts the capabilities of attackers when exploiting existing vulnerabilities. Our evaluation shows that this strict implementation of access control induces no observable performance loss when executing a set of real-world applications. We reported our findings to Oracle and engage in a continuous exchange to investigate this matter further, possibly leading to future changes of the platform. Oracle supports this ongoing work with a dedicated research grant.

1.7 General related work

Each main chapter of this thesis presents related work that specifically concerns its respective contents. Moreover, in Section 6.2 we revisit the contributions of this thesis and discuss their applicability to languages and systems similar to Java.

In this section, we present prior work that is generally related to the topics discussed in this thesis, but has a weaker connection to the techniques or concepts developed and discussed in this work.

Detecting malicious JavaScript code
Browsers retrieve previously unknown JavaScript code for execution from potentially untrusted sources, which involves security-related challenges that also concern Java.

Malicious JavaScript code that attempts to exploit memory corruption vulnerabilities typically contains malicious shellcode to implement its attack. Hence, Egele et al. [30] presented a solution that they integrated into the browser to detect shellcode in JavaScript string buffers to prevent the execution of malicious native code. To this end, they use x86 instruction emulation to check relevant buffers before any code the may contain is executed. Their evaluation shows that this solution is effective while presenting no false findings.

Rieck et al. [93] presented Cujo, a system that can be integrated into a web proxy to transparently analyze web pages requested by a browser to detect malicious JavaScript code before it enters the browser's engine for execution. To this end, Cujo automatically extracts relevant code features and applies machine learning techniques just-in-time to detect malicious patterns in code. In their evaluation, the authors show that Cujo is highly effective and efficient at the same time—it detects 94% of all attacks in a sample set with a low number of false results, while taking a median of 500 ms for analyzing a web page. Cova et al. presented a similar

concept for the same purpose [24]. Although these solutions have been implemented to detect malicious JavaScript code, it might be possible to adapt them to Java to prevent the loading and execution of malicious code in the JVM.

Flash-based vulnerabilities and malware analysis

Flash and Java have in common that they were both integrated as browser plugins for the execution of untrusted code on a large number of workstations.

In the context of Flash-based malware, Wressnegger and Rieck [110] studied the detection rates of malware analysis solutions over time, and also discussed reasons for why certain malicious samples were undetected by scanners at first sight. The basis for their analysis is a sample set of 2.3 million unique Flash animations that the authors collected from VirusTotal until January 2017 over the course of three years, containing malicious and benign samples. As they explain, Adobe Flash has been replaced in large parts by novel browser technologies, but their findings suggest that Flash-based malware is still common today.

Ford et al. [33] developed a solution to detect malicious Flash files by combining static and dynamic analysis. The dynamic analysis component uses Gnash to create an execution trace for the sample under investigation. The data collected statically and dynamically is used for classification. Samples are classified as malicious if they automatically redirect the user's browser to external sites without user interaction, if they contain exploit code for a specific vulnerability, or if they contain shellcode, for example. An evaluation shows that this solution is effective in detecting malicious samples, but the authors also discuss weaknesses and limitations of their approach.

Wressnegger et al. [111] presented Gordon, an alternative approach for detecting Flash-based malware by combining static and dynamic analysis. The dynamic analysis component observes the behavior of a sample and manipulates its control flow to increase coverage in specific regions of the code. Then, based on the data collected statically and dynamically, a linear support vector machine is used to classify samples as benign or malicious. The evaluation shows that the detection rate of this approach is 90% or higher on a sample set of Flash files comprised of ca. 2.000 malicious and ca. 25.000 benign samples.

Runtime and browser security

The .NET framework is a runtime that shares certain characteristics and security challenges with the Java runtime. Desmet et al. [29] studied security policy enforcement for applications executed on mobile devices based on Windows CE and the .NET Compact Framework. As the authors explain, this edition of the .NET framework is functionally limited in comparison to the full version of the framework and hence provides no specific defense mechanisms to securely contain the execution of untrusted code. To address this problem, Desmet et al. discuss the concept of security-by-contract and present an implementation for the .NET Compact Framework that is capable of dynamically enforcing security policies for mobile applications. To this end, the solution instruments all call sites in the application to security-sensitive APIs of the framework so that an execution monitor is asked for permission. The execution monitor keeps track of the program state and decides based on the security policy whether calling the API shall be granted

or denied. While this system is conceptually similar to Java's approach to policy enforcement, it is significantly simpler and less flexible than Java's security architecture.

Grier et al. [45] presented the OP Web browser which combines security-related concepts from operating systems and formal methods to achieve specific security guarantees. Its design comprises five subsystems that are isolated on the level of operating system processes. Moreover, each instance of a browser plugin runs within a separate process. The browser kernel is the central subsystem that is also responsible for access control. For example, the communication between the HTML parsing and rendering component and instances of browser plugins is routed through the browser kernel, which can restrict plugins from accessing certain resources. A design that leverages isolation on the operating system level can potentially reduce the impact of, e.g., remote code execution vulnerabilities in browser plugins like the Java browser plugin.

1.8 Thesis organization

This thesis is organized as follows. Chapter 2 presents technical background information on Java which aids the understanding of the subsequent chapters. Chapter 3 presents an in-depth study of a broad set of Java exploits that serves as a detailed overview of how Java security fails in practice. In Chapter 4 we address that information hiding is a major weak spot of Java's security architecture and present novel approaches on strengthening the runtime such that it withstands common attacks. Chapter 5 takes a different perspective and focuses on increasing security code maintainability. We show that the Java runtime's code maintainability is impaired by hardcoded shortcuts that, under certain circumstances, bypass Java's general principle of stack-based access control. Through empirical analysis we show that these shortcuts can be removed to lower the risk that simple code refactorings introduce security vulnerabilities while, at the same time, retaining high performance. Finally, Chapter 6 concludes.

2

The Java security architecture

This thesis studies the Java security architecture in depth. Understanding the contents of the later chapters requires technical knowledge of the Java platform and its security mechanisms. In this chapter we thus provide the necessary background information and introduce the concepts that will be referred to in the remainder of this work.

2.1 Architectural overview

The following sections explain the technical basics and fundamental concepts of the Java runtime's system architecture.

2.1.1 System scope

The Java Runtime Environment (JRE) is a user-mode software that executes Java applications on a host system. Java applications are implemented in the Java programming language and they are specifically compiled for execution on the Java runtime. Every application is composed of multiple Java classes, and each class is compiled to one or more Java class files. Multiple class files can optionally be bundled in a JAR (Java Archive) file for easier deployment. Each Java application has a defined entry point, the main method. Class files contain Java bytecode which is a platform independent intermediate representation of program behavior. The bytecode instruction set is different to the instruction sets of common processor architectures, so the JRE is required to interpret or compile bytecode in order to execute Java applications.

Figure 2.1 illustrates the system context of the JRE. For program execution, the JRE takes a set of user-provided Java classes as input that collectively resemble a Java application. While executing the application's bytecode, the JRE serves as a mediator between application classes and the host operating system. The JRE itself is implemented in native code, compiled in a platform-dependent way such that the underlying operating system can execute it. Similar to other applications implemented in native code, like "Native Application A" in Figure 2.1, it directly accesses the operating system's extensive service API to implement commonly used features like file system or network access.

Further, Figure 2.1 shows the different ways on how Java code can dynamically enter the system. "Java Application A" represents a case in which the user downloaded and stored the application locally for program execution. As can be seen, it dynamically loads an external class

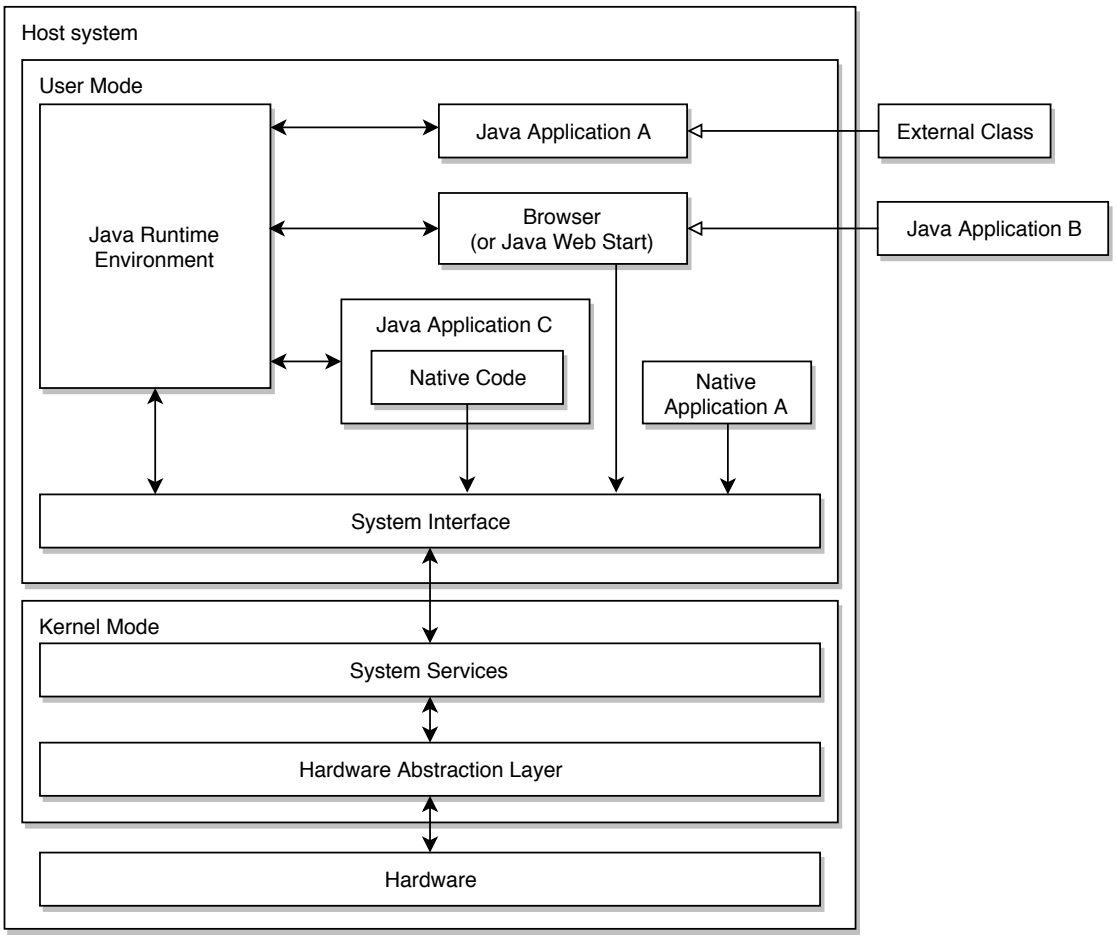


Figure 2.1: System context of the Java Runtime Environment

over a network connection by using built-in functionality of the JRE during program execution. An example use case for such an application design would be a plugin-based product that allows users to dynamically install additional plugins during runtime. Another possible use case is the implementation of a browser in Java that provides functionality for dynamically downloading and executing applets, similar to HotJava.

"Java Application B" shows a different scenario, in which the entire application is downloaded over a network connection by a browser plugin, or Java Web Start. In this case, the application is merely cached on the host machine and then provided to the JRE for program execution.

Figure 2.1 shows another important aspect of the system context in the example of "Java Application C". It is possible for Java applications to use natively compiled libraries which have direct access to the system interface. These libraries need to be specifically prepared and compiled for use by Java classes.

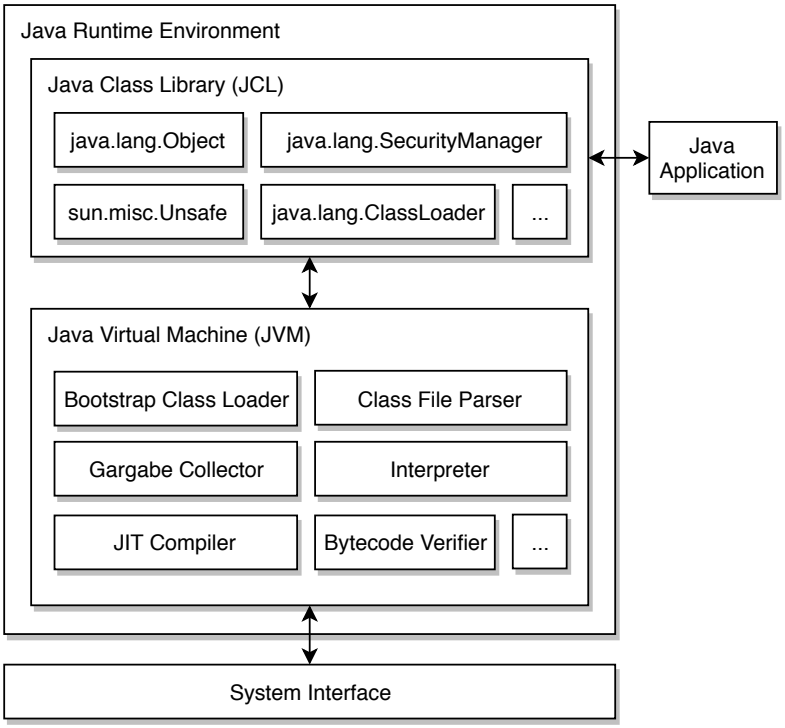


Figure 2.2: High-level system architecture of the JRE

In summary, one can see that there are various different ways on how Java applications can be designed. Moreover, it can be seen that the JRE is specifically designed to support portable and distributed applications.

2.1.2 High-level components

As shown in Figure 2.2, the JRE comprises two high-level components. The first component is the Java Virtual Machine (JVM). It is responsible for executing the bytecode contained in Java class files. For this, it implements various different features, including a class file parser, an interpreter, just-in-time (JIT) compiler, a garbage collector for memory management, and more. It is implemented in native code and thus platform dependent.

The second high-level component is the Java Class Library (JCL), which is a large set of system classes that implement commonly used features, such as user interface functionality, file access, or network access. In that sense, the JCL is similar to the standard libraries of other languages, such as, for example, the libc. Most parts of the JCL are implemented in Java, so just as for application classes the JVM is responsible for their execution. Application classes can use the features implemented in the JCL by means of method calls, class inheritance, callbacks, etc. By design, any meaningful Java code is required to call methods in the JCL, because the bytecode instruction set does not provide any commands for direct access to native libraries or operating system features.

Listing 2.1: Java class using a native method

```
1  public class Main {
2    static {
3      System.loadLibrary("Main");
4    }
5
6    public static void main(String... args) {
7      System.out.println(reverseText("Hello, world!"));
8    }
9
10   private static native String reverseText(String text);
11 }
```

System classes interact with the JVM to implement low-level features, such as memory access, and the JVM in turn accesses the operating system's service API. The natively implemented JVM takes an intermediary role between the JCL and the operating system's API. The JCL is mostly platform independent and thus provides the same public interfaces for use by applications on different operating systems. Due to this, Java applications that do not contain any native libraries need to be compiled only once to be executed on different operating systems.

2.1.3 The Java Native Interface

As already indicated in Figure 2.1, it is optionally possible for system classes and Java applications to provide their own native code. The Java Native Interface (JNI) is an API which, as its name suggests, interfaces between Java methods and native functions. Specifically, JNI allows Java methods to call functions implemented in native libraries, and it allows native code to call Java methods. Calls between native code and Java classes can exchange arguments and return values just like regular method or function calls. JNI can only be used to call native functions in libraries that follow a specific naming scheme, it is thus not possible to call functions in arbitrary libraries.

For illustration purposes, consider Listing 2.1. In this example, `Main` is a Java class that contains three elements. In lines 2–4 is a static initializer that will be automatically executed by the runtime when `Main` is initialized, i.e., before any of its methods or constructors are executed. It contains a call to `System.loadLibrary()` that will cause the runtime to dynamically load a native library referred to as "Main". The second element in lines 6–8 is the `main()` method, which is the application's entry point. The runtime will call this method after initializing the class, i.e., after the static initializer has been executed. The third element in line 10 is a definition of a native method `reverseText()` which indicates the runtime that there is a native library that implements a function that matches the specified signature. Java classes do not contain method bodies for native methods, so before Java code can call a native method, the native library that implements the method must be loaded first. In the example, this is done in the static initializer by the call to `System.loadLibrary()`. Also, the native library must follow a specific naming scheme to ensure that the Java runtime is able to correctly match the native function to the method

Listing 2.2: Implementation of a native method in C++

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include "Main.h"
4
5  JNIEXPORT jstring JNICALL Java_Main_reverseText
6    (JNIEnv *env, jclass, jstring text) {
7    const char *chars = env->GetStringUTFChars(text, NULL);
8    int strLen = strlen(chars);
9    char *reversed = (char*)malloc(strLen);
10   for(int i = 0; i < strLen; i++) {
11     reversed[i] = chars[strLen-i-1];
12   }
13
14   return env->NewStringUTF(reversed);
15 }
```

definition in the Java class. This is also the reason for why Java code cannot use functions in arbitrary native libraries.

Listing 2.2 shows the corresponding C++ code for the native library that implements `reverseText()`. As can be seen, the native code in this listing makes heavy use of JNI types and functions, which is required to allow for the interaction between Java code and C++ code. It imports "Main.h" in line 3, which is a header file that was automatically generated by "javah", a helper application that is supplied with the Java Development Kit (JDK). It takes as input any given Java class and it outputs a C-style header file that already includes all structures and definitions that are needed for interaction with the supplied Java class. The generated header file, in turn, includes "jni.h", which is also part of the JDK. Among other things, it defines various C types that correspond to the different Java types, such as `jclass` and `jstring`.

Lines 5–6 contain the function header of `reverseText()` which deviates from the method signature defined in `Main`. One difference is the function name, which in the C++ code is a fully-qualified reference to the Java method, including the class and method name. This is required so that the Java runtime can match the native function to the corresponding Java method. Another difference concerns the list of parameters. The C++ function has an additional parameter `env`, which points to a structure that contains pointers to JNI functions. The second additional parameter refers to the Java class that defines the native method, in the example this is `Main`. The third parameter represents the actual parameter `text` of the Java method. Another noticeable difference is that JNI automatically maps Java types to equivalent platform-dependent native types, in the example this concerns the return value and the parameter `text` which are both of type `jstring`.

Line 7 shows how the native code uses `env` to call `GetStringUTFChars()` to convert the string of type `jstring` to a C-style character array. The code in lines 8–12 subsequently performs the required actions to reverse the string that was supplied as an argument by the Java method. Line 14 finally returns the reversed string to the Java method that called `reverseText()`. For

this, it calls `NewStringUTF()` to convert the C-style character array to a `jstring`, which is the equivalent to an instance of `java.lang.String`. Due to the fact that JNI automatically maps Java types to platform-dependent native types, and vice versa, calling a native method in Java is no different to calling a regular Java method, as can be seen in Listing 2.1 in line 7. The application in the example will output "!dlrow ,olleH".

2.1.4 Class loading

Class loading is the mechanism that initially loads Java classes into the runtime so that they can be executed by the JVM. The following introduces the basic concepts of Java's class loading process.

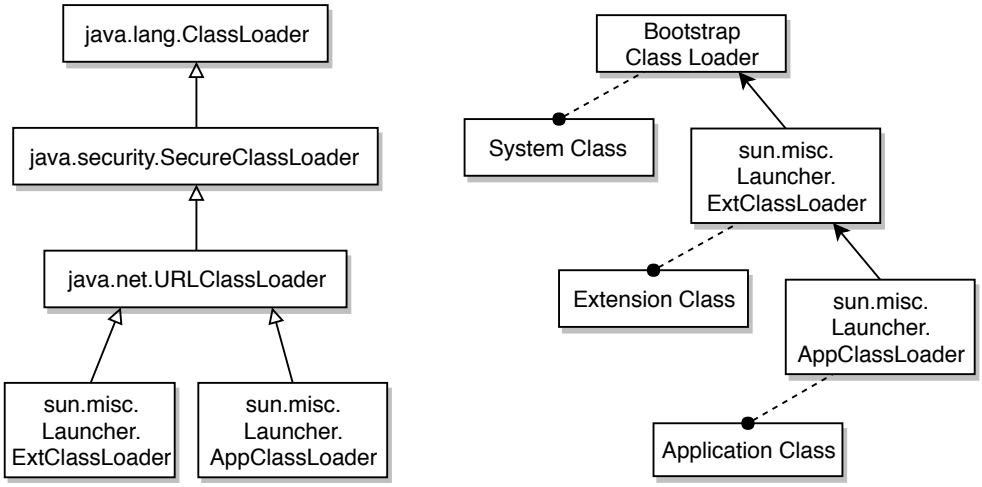
Class hierarchy of loaders

The JRE provides a set of different class loaders, and each class loader has different responsibilities and capabilities. The *bootstrap class loader*, also referred to as the "primordial class loader" or "null class loader" [37], is responsible for loading system classes that belong to the JCL, such as `java.lang.Class` or `java.lang.Object`. It is part of the JVM and implemented in native code, and therefore not represented as a Java class [37]. It searches for classes in specific folders, which we will refer to as the *boot class path*. One of the many system classes that are loaded by the bootstrap class loader is `java.lang.ClassLoader`, an abstract class that all other class loaders except the bootstrap loader need to subclass, either directly, or indirectly. An example of a concrete class loader is `java.net.URLClassLoader`, which in turn serves as a superclass for `sun.misc.Launcher.ExtClassLoader` and `sun.misc.Launcher.AppClassLoader`.

`ExtClassLoader` is the *extension class loader*, which loads all classes that belong to extensions, i.e., packages that do not belong to the JCL, but extend the platform in an application-independent manner. One example of an extension is the SunJCE, a set of cryptographic providers that for historical reasons were not included into the JCL. Extensions are located in dedicated folders that are known to the platform, which we call the *extension class path*. `AppClassLoader` is the *application class loader*, which is responsible for loading application classes. All application classes need to be stored in folders that belong to the *application class path* so that they can be found by the application class loader.

Besides the class loaders discussed above, application developers are free to implement their own custom class loaders by extending directly from `java.lang.ClassLoader`, or any of its subclasses. These custom class loaders can be used to implement application-specific class loading features not provided by one of the standard loaders, such as dynamic class generation, or loading encrypted code [62].

Figure 2.3a shows an overview of the class hierarchy of different class loaders, with arrows indicating inheritance. As can be seen, all loaders eventually subclass `java.lang.ClassLoader`. The bootstrap class loader is not included in this illustration due to the fact that it is not represented as a Java class and thus not involved in any class inheritance relationships. Also note that this illustration is incomplete, because there are additional class loaders shipped with the JRE that we have not discussed, as well as application-specific class loaders deployed with their respective applications.



(a) Class hierarchy of class loaders (b) Delegation hierarchy and class definition relationships

Figure 2.3: Hierarchical structures of class loaders

Loader delegation

Converting a binary representation of a class into an instance of `java.lang.Class` is called *class definition*. The instance of a class loader that defined a certain class is called its *defining loader*. The type of any given class is defined by the tuple ⟨*bn, dl*⟩ [62, Sec. 5.3], with *bn* representing the binary name of the class [62, Sec. 4.2.1], which is a fully qualified name of the class as it appears in the class file, and *dl* being the class loader that defined the class. Classes that were defined by the bootstrap loader will report `null` as their defining loader, because the bootstrap loader is implemented natively and not represented as a Java class. This concerns all system classes, as the bootstrap loader is responsible for loading all classes that belong to the JCL. For example, the types of the system classes `java.lang.Object`, `javax.crypto.Cipher` and `java.io.PrintStream` are ⟨*java/lang/Object, null*⟩, ⟨*javax/crypto/Cipher, null*⟩ and ⟨*java/io/PrintStream, null*⟩, respectively.

Due to this definition of class types, it is important to ensure that that all classes are defined by appropriate loaders to avoid compatibility issues. For example, if an application implements its own custom class loader that also defines system classes, like ⟨*java/lang/String, CustomClassLoader*⟩, it will be incompatible with other system classes that return values of the type ⟨*java/lang/String, null*⟩. Although these two types may have been created based on the same binary representation for `java.lang.String`, they are considered different by the runtime.

To avoid this problem, all class loaders that are shipped with the Java runtime follow the *class loader delegation model*. In general, loader delegation allows class loaders to either load specific classes themselves, or request other class loaders to do so, which may continue recursively until eventually one loader defines the requested class, or decides that the requested class cannot be found. The loader that originally initiated the loading process for a specific class is called the *initiating loader* or *requesting loader*. To ensure a deterministic process, all class loaders are organized in a hierarchical manner, with each loader having exactly one parent loader, and

zero or more children. As illustrated in Figure2.3b, this structure is not related to the class hierarchy of class loaders, which is shown in Figure 2.3a. For example, `ExtClassLoader` and `AppClassLoader` are both direct subclasses of `URLClassLoader`, but `ExtClassLoader` is the delegation parent of `AppClassLoader`. The class resolution algorithm [37] determines the delegation process as follows:

1. The initiating loader first checks whether the requested class has already been loaded. If this is the case, the loader must return the previously returned instance of `java.lang.Class` [62].
2. If the class has not been loaded before, the loader delegates to the parent class loader. If the parent loader is able to find the class, return the instance it returns.
3. If the parent class loader has not found the requested class, try to load the class.

As an example, consider again the application in Listing 2.1. Class `Main` refers to `java.lang.System`, which eventually will cause the runtime to load this class when executing the application. For this, the runtime will request `System` from the same loader that loaded `Main`. In our example, we will assume that this is Java's application class loader, `sun.misc.Launcher.AppClassLoader`. We further assume that `System` has not been loaded before. The application class loader follows the class resolution algorithm and delegates the loading of `System` to its parent loader, which is the extension class loader, `sun.misc.Launcher.ExtClassLoader`. The extension class loader, in turn, delegates to the bootstrap class loader. The bootstrap loader is able to locate the binary representation for `System` on the boot class path and thus returns an instance of `Class` with the type ⟨*java/lang/System, null*⟩. This instance of class will propagate all the way up to the application class loader, which originally initiated the loading. Subsequent requests to load `System` will terminate early and, following the class resolution algorithm, always return the same instance of `Class` with type ⟨*java/lang/System, null*⟩.

Figure 2.3b provides an overview of the delegation hierarchy for the different loaders. The fact that all loaders search for classes only in their respective class paths ensures that all classes are defined by their appropriate loaders—system classes are defined by the bootstrap class loader, extension classes by the extension loader, and application classes by the application class loader.

2.1.5 Class introspection

In general, *reflection* in computer programming refers to the ability of code to inspect or modify its own representation. This concept is widely supported by modern programming languages such as C#, Go and PHP.

In Java, the reflection API and MethodHandles can be used to inspect the structure of classes, i.e., retrieving the set of variables and methods contained in a class. It is also possible to read and write the values of variables, invoke methods, or instantiate objects. Reflection allows for access to class members during program execution that were not known or defined at compile time. It is thus possible to dynamically specify the fields or methods that shall be accessed by a class using information gathered during runtime. A possible use case for this functionality is object serialization. The serializer can use reflection to inspect the object that shall be converted to a byte array, and then read all its variables and output them to disk. A serializer that is implemented

Listing 2.3: Example use of the reflection API

```
1  import java.lang.reflect.Field;
2
3  public class Main {
4
5    private String myString = "Example";
6    private int myInt = 7;
7    private double myDouble = 3.14;
8
9    public static void main(String[] args) throws Throwable {
10     printObjValues(new Main());
11   }
12
13   public static void printObjValues(Object o) throws Throwable {
14     Field[] fields = o.getClass().getDeclaredFields();
15     for (Field f : fields) {
16       System.out.println(f + " = " + f.get(o).toString());
17     }
18   }
19
20 }
```

this way is capable of converting arbitrary types because it uses reflection for field access, rather than hardcoding all variables of a set of well-known types.

For illustrating reflective access to field values, consider the example in Listing 2.3. The method `printObjValues()` accepts an arbitrary object as argument, and it prints all declared fields of this object along with their respective values. For this, it first uses the reflection API to retrieve the list of all fields of the object in line 14. Then, it retrieves the field value for each field in line 16 and prints it. The output of the application is:

```
private java.lang.String Main.myString = Example
private int Main.myInt = 7
private double Main.myDouble = 3.14
```

Although the reflection API and MethodHandles provide broad support for class introspection, there are fundamental restrictions. Java provides no support for self-modifying code, which has two important advantages. First, it avoids conceptual complexity that would otherwise be required to support dynamic code modification. After all, the Java source code of an application is much different from its bytecode representation in terms of notation and the paradigms it uses, which in turn is much different to its machine code representation. An API for self-modifying code would need to bridge this gap in such a way that modifications of one method would not require dynamic re-evaluation and recompilation of the entire application, while being easy to use at the same time. Second, as we will explain in more detail later, there are certain bene-

fits in terms of security when avoiding self-modifying code. Specifically, code can be verified once before its first execution [62, Sec. 4.10], and the security guarantees achieved through this evaluation hold throughout program execution.

Java not only prevents self-modifying code, it also prevents the introspection of method bodies that were loaded into the runtime. Although executable (and thus analyzable) code is stored in the Java runtime's memory, there is no official API that can be used to access the code. If, for example, code was loaded from an encrypted archive using a custom class loader (see Section 2.1.4), then this code cannot simply be analyzed by Java classes.

2.2 Protection mechanisms

The Java platform is a runtime environment for code potentially obtained from untrusted sources. As explained in Section 2.1.1 there are various ways on how external code can enter the host system. Besides many advantages, this openness also entails various security risks. Attackers may want to illegitimately access private data stored on the host machine or abuse it for illegal activities. The Java runtime implements a security architecture that comprises various different protection mechanisms in order to mitigate these threats and prevent damage to the host system. The foundation of how these different mechanisms are supposed to work and interplay is defined by the Java security model which was extensively described by Gong et al. [39]. Note that the scope of the Java runtime's protection mechanisms is limited to Java code only. Any native code that is used by Java applications is by no means restricted by the runtime. However, as explained below in detail, the runtime can prevent untrusted Java applications from loading and using native code in the first place.

The following introduces conceptual and technical details of the Java security architecture which is required background information for the subsequent chapters. Unless otherwise stated, the information provided below is based on the work of Gong et al. [39], to which we refer the reader for more in-depth details.

2.2.1 Stack-based access control

A central protection mechanism of Java's security architecture are permission checks by means of stack-based access control. This mechanism limits access to certain functionalities of the JCL which restricts the capabilities of code to prevent undesired activities. The following introduces important primitives related to access control.

Security policy

Users and administrators of the Java runtime can set customizable security policies that assign access permissions to code. This can be done either programmatically, or by supplying a policy file. Permissions can be assigned in a fine-grained manner thus allowing that only parts of an application are granted or denied certain access. The code to which a certain permission shall be granted can be characterized by different properties, including, e.g., its source ("codeBase")

Listing 2.4: Example security policy

```
1  grant signedBy "MyCompany" {
2     permission java.io.FilePermission "C:/temp/*.txt", "read";
3  };
4
5  grant codeBase "file:C:/MyApplication/-" {
6     permission java.io.FilePermission "C:/conf/*", "read,write";
7  };
8
9  grant {
10    permission java.util.PropertyPermission "os.name", "read";
11 };
```

and digital signature ("signedBy"). Also, the exact kind of access that shall be granted can in many cases be defined at a detailed level.

For illustration, consider the example policy in Figure 2.4. It contains three assignments. The first assignment in lines 1-3 grants all code that was signed by "MyCompany" read access to files in "C:/temp/" whose name ends with ".txt". The second assignment in lines 5-7 grants all code that resides under the local file path "C:/MyApplication" read and write access to all files in "C:/conf/". Finally, the third assignment in lines 9-11 contains no specific characteristics and thus grants all code read access to Java's system property "os.name". Note that more than one assignment can be applicable to code if it exhibits the specified characteristics in which case all concerning permissions are granted additively. For example, in the context of Listing 2.4 this would mean that all code that resides in "C:/MyApplication" which is also signed by "MyCompany" is granted all three permissions.

Security manager

The security manager is responsible for security policy enforcement. Java's standard implementation of the security manager is represented by the class `java.lang.SecurityManager`. By default, no security manager is in effect when program execution starts. In this case, no security checks will be performed and the security policy will thus not be enforced, i.e., all code has unlimited access to the JCL and the host system. Putting a security manager in charge can be done programmatically by the application by calling `System.setSecurity-Manger(SecurityManager sm)`, or by providing an appropriate command line argument to the JVM. In any case, the instance of `SecurityManager` that is in charge of policy enforcement is stored in the private field `System.security`. If policy enforcement is disabled, this field contains `null`.

Figure 2.5 shows how `File.createNewFile()` consults the security manager to implement a security check. It first calls `System.getSecurityManager()` in line 3 to retrieve the currently active security manager. If this call returns `null`, i.e., policy enforcement is disabled, the security

Listing 2.5: Example of a permission check in `java.io.File`

```
1  ...
2    public boolean createNewFile() throws IOException {
3      SecurityManager security = System.getSecurityManager();
4      if (security != null) security.checkWrite(path);
5      if (isInvalid()) {
6        throw new IOException("Invalid file path");
7      }
8      return fs.createFileExclusively(path);
9    }
10 ...
```

check will be skipped. Otherwise, `SecurityManager.checkWrite()` is called in line 4 to check if the calling code is allowed to write to the requested destination path. The security manager will return silently from the security check if access is granted, and throw an exception otherwise. Hence, `fs.createFileExclusively()` in line 8 will only be reached if either policy enforcement is disabled, or the calling code has appropriate permissions to write to the specified folder. This example in `java.io.File` is representative for how the JCL protects access to sensitive functionality.

Permissions

The previous sections explained that security policies can be used to assign permissions to code. The JCL contains a standard set of permissions that can be used to specify policies. All permissions are typed and optionally parameterized. Table 2.1 provides an incomplete overview of the permissions that are contained in the JCL. All permissions are represented as Java classes that share the common abstract superclass `java.security.Permission`. As can be seen from the table, applications require specific permissions, e.g., when attempting to load native libraries, when trying to reflectively access class members that would otherwise be protected by information hiding, or when setting a security manager. Listing 2.4, for example, assigns permissions of types `FilePermission` and `PropertyPermission` with actual parameters that specify the exact kind of access that shall be granted.

Permissions can imply each other, e.g., access to "C:/*.*" implies access to "C:/input.txt". For this, the different types of permissions must implement appropriate decision logic in their respective `implies()` methods to ensure proper and meaningful access control checks.

Protection domains

The security policy assigns permissions to code, but as we will explain in this section, in the Java runtime's internal representation these permissions are not directly assigned to classes and objects. Instead, the runtime assigns classes and their corresponding objects to protection domains, e.g., based on their code source or digital signature, and then assigns permissions to

Table 2.1: Subset of Java's standard permissions

Permission	Comment
`java.security.AllPermission`	Grants all permissions and thus unlimited access to features of the JCL.
`java.io.FilePermission`	Grants access to the file system. Arguments specify allowed paths and allowed actions, including "read", "write", "execute", and "delete".
`java.security.SecurityPermission`	Grants access to certain security-critical features. Parameters specify the exact kind of access that shall be granted, e.g., the "setPolicy" argument allows code to set the security policy.
`java.net.SocketPermission`	Grants access to network connections via sockets. Arguments can be used to specify allowed hosts, e.g., by URL, and allowed ways to connect to the host, including "accept", "connect", "listen", and "resolve".
`javax.sound.sampled.AudioPermission`	Grants access to audio resources. Arguments can be "play" or "record".
`java.lang.reflect.ReflectPermission`	Allows for bypassing Java's rules for information hiding when reflectively accessing class members, e.g., by calling `AccessibleObject.setAccessible()`
`java.lang.RuntimePermission`	Covers a broad range of different actions, arguments specify the exact kind of access that shall be granted. For example, "loadLibrary.{library name}" allows for dynamically loading the specified library, and "setSecurityManager" allows for putting a security manager in charge of policy enforcement.

these protection domains. So effectively, all classes are granted the permissions that are assigned to the protection domain to which they belong. The runtime manages classes, protection domains, and their respective permissions in such a way that they exactly reflect the security policy specified by the user.

Listing 2.6 shows the `defineClass()` method of `SecureClassLoader` to illustrate how the runtime assigns classes to protection domains when they are first defined. This is the default behavior of all standard class loaders because they all subclass `SecureClassLoader`, as can be seen in Figure 2.3a. Note that each class is assigned to exactly one protection domain, but every protection domain can hold more than one class.

Protection domains serve as a level of indirection that is used to group together classes that belong to the same trust boundary. Such a clustering within the Java runtime's internals facilitates class management and access control decisions—rather than performing access control checks

Listing 2.6: `SecureClassLoader` assigns new classes to protection domains

```
1  ...
2    protected final Class<?> defineClass(String name, byte[] b, int
         off, int len, CodeSource cs) {
3      return defineClass(name, b, off, len, getProtectionDomain(cs)
           );
4    }
5  ...
```

on the basis of individual classes and objects, the runtime can perform checks on the level of domains. If, for instance, all code that is involved in a specific call sequence belongs to the same protection domain, and this domain was granted all permissions, then any permission checks that might occur in that sequence can be skipped safely.

All classes that belong to the JCL were loaded by the bootstrap loader and belong to the same protection domain. This means that all system classes are considered equally trustworthy and that they all share the same set of privileges. By default, all system classes are granted all permissions, i.e., they have unlimited access to each other and the host system that executes the Java runtime.

Permission checks

The Java runtime enforces the security policy by means of permission checks. To this end, all security sensitive methods in the JCL are guarded by calls to the security manager, thus ensuring that permission checks are executed before any sensitive action is performed. Listing 2.5 shows a representative example on how system classes interact with the security manager.

Java's approach to checking permissions takes into account that the execution of an application typically involves call sequences that traverse different protection domains. For example, an application will at some point call methods of the JCL to write data to disk. Assuming that the application belongs to a lower privileged protection domain than the JCL, such a call would traverse a boundary. In other cases, code in the system domain will call methods in application classes. Graphical user interface components in Java, for example, will invoke callback methods in application classes in order to pass events, such as mouse clicks. Also these calls would traverse a trust boundary between different protection domains. In all these cases, it is important to ensure that privileges are not accidentally transfered from a higher privileged domain to a lower privileged domain, i.e., application classes with less privileges than system classes must not gain additional permissions by calling system methods, or by being called by a system method.

Java's permission checks were specifically designed to address this concern. Whenever a permission check is triggered by a call to the security manager, the runtime inspects the call stack of the executing thread to determine the set of all protection domains of all callers on the stack. If and only if the requested permission was granted to each and every protection domain in this set, the permission check succeeds, see Figure 2.4. Otherwise, the permission check fails,

$$p \in \left\{ \bigcap_{i=1}^{n} D_i \right\}$$

Figure 2.4: Set notation of permission checks as defined by Gong et al. [39, p. 95]. A permission check succeeds if and only if the requested permission p is an element of the intersection of the permissions of all n protection domains D.

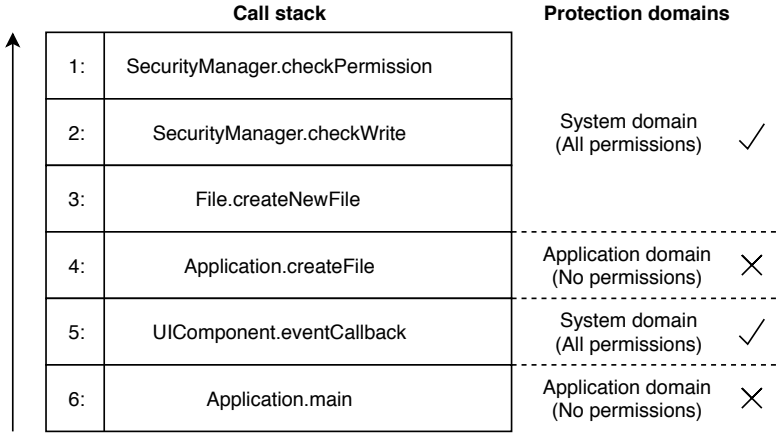


Figure 2.5: Hypothetical call stack of a call sequence that attempts to create a file

because at least one of the callers on the stack belongs to a protection domain with insufficient privileges. This approach to access control ensures that code in higher privileged domains cannot accidentally transfer privileges to code in lower privileged domains just by performing method calls that traverse domains.

Implementing stack-based access control as described above is one of the major changes that were introduced with the release of Java 1.2. This original approach was specifically developed for the Java runtime by Gong et al. [40], but its design was inspired by many different concepts that were previously proposed, e.g., for securing program execution on the Multics operating system.

For illustration, consider the hypothetical call stack of a Java thread in Figure 2.5. The thread was initiated by `Application.main()` which is on the bottom of the stack at position 6. We suppose here that this method created a user interface using system classes, and that one of these system classes, `UIComponent` at position 5, invokes a callback method in the application, which is `Application.createFile()` at position 4 of the stack. This method calls `File.createNewFile()`, which in turn calls `SecurityManager.checkWrite()` to ensure that the callers have been granted the appropriate `FilePermission` that is required to create files in the respective folder. The `checkWrite()` method in `SecurityManager` redirects to the general `checkPermission()` method, which finally triggers a check for the required permission.

As explained above, the permission check first collects the set of all protection domains on the current call stack. In the example, there are two different domains involved: All system

classes belong to the system domain, and the application class belongs to its own domain. Then, the runtime determines the intersection of all permissions of these two domains. The system domain has all permissions by default, but as indicated in Figure 2.5, the application domain has not been assigned the `FilePermission` that is required for creating the requested file. In consequence, also the intersection of the permissions of the two domains does not contain the required permission and the permission check fails with an exception. This example shows that lower-privileged classes cannot simply gain privileges by calling system methods, or by being called by system methods.

Access control context

Permission checks in Java are performed by inspecting a set of protection domains, see Figure 2.4. The set of protection domains that need to be considered for an access control check is referred to as the *access control context*. Typically, a permission check is carried out within the same thread that is trying to access a protected resource, as in the example shown in Figure 2.5. In that case, the access control context is simply the set of all protection domains of all callers on the current call stack.

In some cases, however, it is desirable to carry out an access control check within a different context than the executing thread. A thread pool, for example, can pass custom access control contexts to its worker threads so that they can perform permission checks in individualized contexts that were supplied with different sets of permissions. For these purposes, the class `java.security.AccessControlContext` can be used to encapsulate access control contexts. Instances of `AccessControlContext` hold an array of protection domains and can be passed between different methods and thus between different contexts. The method `AccessControlContext.checkPermission()` can then be used to perform a permission check on the set of protection domains that are encapsulated in the respective instance of `AccessControlContext`, rather than on the currently executing thread.

Privileged blocks

The previous sections explained that a permission check considers an entire access control context which contains one or more protection domains involved in a specific call sequence. In the typical case, this access control context contains all protection domains of all callers on the current call stack. The purpose of this is to ensure that all callers were assigned sufficient privileges before executing a sensitive operation. In specific situations, however, this behavior is not desired, i.e., a security-sensitive action is supposed to be executed even though not all callers on the stack have been assigned the required permissions.

There are various use cases in which it would be advantageous for high-privileged classes to restrictively execute sensitive actions on behalf of unprivileged classes. The system class `java.security.SecureRandom` is a real-world example for this. This class can be used to generate strong random numbers for cryptographic purposes. Its public method `getInstanceStrong()` can be used to retrieve an instance of `SecureRandom` that implements an algorithm for random number generation that was specified in the security property "securerandom.strongAlgorithms". This means that when being called the method `getInstanceStrong()`

Listing 2.7: Example of a privileged block in `java.security.SecureRandom`

```
1   ...
2   public static SecureRandom getInstanceStrong() throws
        NoSuchAlgorithmException {
3     String property = AccessController.doPrivileged(
4     new PrivilegedAction<String>() {
5       @Override
6       public String run() {
7         return Security.getProperty(
8         "securerandom.strongAlgorithms");
9       }
10    });
11    ...
12  }
13  ...
```

must read the security property's value so that it can create and return the correct instance of `SecureRandom`. Note that reading values of security properties is considered a security-sensitive operation in Java's security model, which is why a `SecurityPermission` is required for this. If `getInstanceStrong()` would simply call `Security.getProperty()` to read the property when being called, then `getProperty()` would trigger a permission check that would inspect all callers on the stack to ensure that they were granted the required `SecurityPermission` for property access. As a consequence, if unprivileged application classes were to call `getInstanceStrong()`, this permission check would fail and the application would be incapable of retrieving the desired instance of `SecureRandom`.

But granting untrusted code the permission to access security properties just to provide access to strong random number generators is not an option, as this would violate the principle of least privilege. Instead, in this case it would be beneficial if `SecureRandom` could read the security property without requiring its own callers to have the same privilege. The solution to this problem is the Java runtime's API for privileged blocks. Java classes can temporarily elevate the privileges of other callers on the stack by calling `AccessController.doPrivileged()`, which allows the execution of sensitive operations even if not all callers were granted the required privileges. Using the privileged block API is a way for system classes to vouch for the secure use of sensitive functionality even if the system class's methods were called from untrusted code.

Listing 2.7 shows how `SecureRandom` uses a privileged block to access a security property. For this, it calls `doPrivileged()` in line 3. As an argument, it supplies an anonymous instance of `PrivilegedAction` which is defined between lines 4 to 10. When calling `doPrivileged()`, the runtime will call the `run()` method of this privileged action, which in turn contains the actual code that is supposed to be executed with elevated privileges. In this example, the `run()` method contains the call to `Security.getProperty()`.

The consequence of using the privileged block API in the example is that now even unprivileged application code is able to call `getInstanceStrong()` without causing any permission

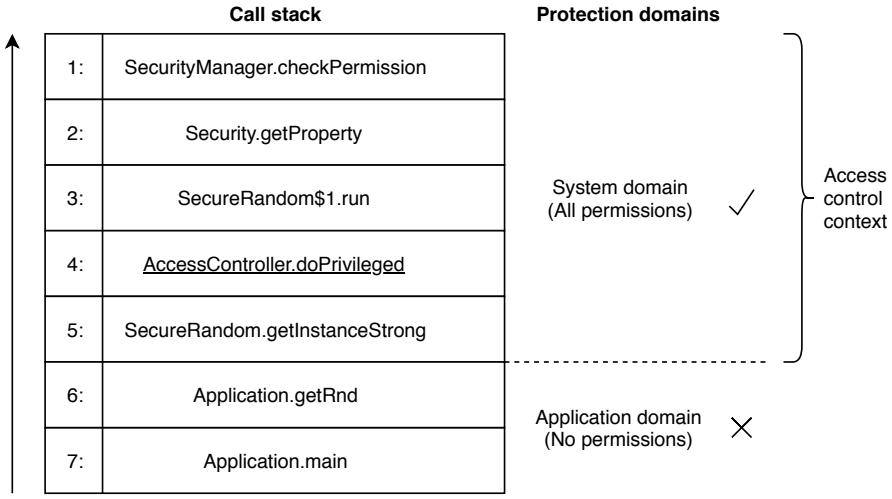


Figure 2.6: Call stack of an application that uses `SecureRandom`

checks to fail. Note that the permission check in `Security.getProperty()` is still being executed. The difference is, however, that the use of `doPrivileged()` will influence the access control decision by reducing the set of protection domains that will be considered for the permission check. Consider the call stack in Figure 2.6. If a basic approach to permission checking were to be applied that would be unaware of the special semantics of `doPrivileged()`, then the runtime would first collect the set of protection domains of *all* callers on the current call stack. In the example, this would be the system domain that holds all system classes, and the application domain that holds `Application`. The permission check triggered by `Security.getProperty()` would then depend on the permissions assigned to the application domain, as system classes are already granted all permissions by default. An application without the required `SecurityPermission` would cause the permission check to fail. Instead of this basic approach, however, the runtime will apply an algorithm for making access control decisions similar to the pseudocode shown in Listing 2.8. This approach takes into account the special semantics of `doPrivileged()`, and the runtime will only check the privileges of the caller of `doPrivileged()` and the subsequent callers on the stack. It will, however, ignore all callers that preceded the caller of `doPrivileged()` on the call stack. For the example call stack shown in Figure 2.6, this means that `SecureRandom.getInstanceStrong()` and all subsequent calls will be considered for the permission check, but `Application.getRnd()` and `Application.main()` will be ignored. As a consequence, even unprivileged application classes can call `getInstanceStrong()` without causing the permission check in `getProperty()` to fail. Note that the approach to permission checking illustrated in Figure 2.4 is still applied, the privileged block API will merely reduce the set of protection domains that belong to the access control context considered for the check.

Listing 2.8: Pseudocode of the access control algorithm used for permission checks

```
1  markedAsPrivileged = false;
2  for(i = AccessControlContext.getLength; i > 0; i--) {
3    tempCaller = AccessControlContext.getCallerAtPosition(i);
4    if(!getProtectionDomain(tempCaller).hasRequestedPermission)
5      throw Exception;
6    if(markedAsPrivileged)
7      return;
8    if(isDoPrivileged(tempCaller))
9      markedAsPrivileged = true;
10 }
```

Restricted classes

Java's general approach to access control is that sensitive methods in the JCL utilize the security manager to execute a permission check. An example showing the code idiom that system classes typically use for this is provided in Listing 2.5. Guarding sensitive methods in this way prevents untrusted code from performing security-sensitive operations that could potentially harm the host system. But besides, there is an entire set of special classes that implement sensitive functionality in methods that are not protected by permission checks. These classes reside in so-called *restricted packages*, and the classes they contain are therefore referred to as *restricted classes*. The runtime uses an alternative design to prevent untrusted code from accessing methods in restricted classes, whose rationale and implementation details are not well documented. It is thus reasonable to consider the concept of restricted packages and classes a deviation from Java's security model.

An example of a restricted package structure is `sun.*`, including all packages and subpackages it contains. One well-known example of a restricted class is `sun.awt.SunToolkit`. In Java 7, it contained a method `getField()` that allowed arbitrary callers to reflectively access the contents of arbitrary fields in arbitrary classes. This is considered security-critical functionality, because it bypasses Java's rules for information hiding, which potentially has security-relevant implications. The method itself, however, does not implement a permission check. Instead, the runtime implements a permission check in its application class loader that prevents the loading of classes in restricted packages, unless a specific permission was granted to the application code. Therefore, an exception will be thrown by the application class loader whenever unprivileged application classes reference `SunToolkit`, or any other class that belongs to a restricted package. This is a common way of how the runtime prevents unprivileged code from accessing code in restricted classes.

Besides, some restricted classes implement their own custom ways of preventing access to their methods from unprivileged code without using permission checks. A prominent example of such a class is `sun.misc.Unsafe`. As its name suggests, it provides various methods that can be used to bypass Java's security mechanisms. For example, it allows for unsafe memory operations that would otherwise be prevented by the runtime to prevent harm. Instead of implementing

permission checks, it declared its own constructor to be private so that it cannot be called by other classes. For class instantiation, it implements a method `getUnsafe()`. This method, however, is caller sensitive, i.e., it behaves differently depending on its immediate caller. If the immediate caller was loaded by the bootstrap class loader, it assumes that the access is legitimate, and returns an instance of `Unsafe`. In other cases, i.e., if the immediate caller was loaded from a class loader other than the bootstrap class loader, an exception is thrown. This effectively prevents application classes from retrieving an instance of `Unsafe`, even if they were granted access to restricted packages.

The reasons for why restricted packages were introduced in the first place are not entirely clear. Possibly, this is the result of design decisions made early in the history of Java, and a strong desire to retain backward-compatibility over time. Another possible reason is that waiving permission checks is a potential measure to improve execution speed. After all, each permission check is a rather costly operation as stack inspection is non-trivial. Keeping entire package structures off limits is thus potentially more efficient.

2.2.2 Bytecode verification

The JVM implements a sophisticated approach to static class file verification. One major concern of this is to ensure the well-formedness and sanity of the bytecode contained in the class files that are provided to the JVM for execution. The runtime verifies each class file during class loading before any of its methods are executed. Class files that do not pass the verification will be rejected by the JVM and the code they contain will not be executed. Instead, an exception will be thrown to indicate that verification failed. The Java Virtual Machine Specification stipulates a specific approach for class file verification by type checking [62, Sec. 4.10.1]. For this, the specification defines a type system, a large set of Prolog predicates, and detailed instructions on how to type check classes, methods, and individual instructions. The result of class file verification is a formal proof that all methods contained in a class file conform to the rules specified in the specification [62, Sec. 4.10.1]. The guarantees provided by this includes the following:

- All methods in the class contain only valid instructions. Moreover, all arguments provided to bytecode instructions are valid, i.e., they reference valid data stores or structures in the class file and are of valid types. Instructions that influence the control flow, such as jump instructions, point to valid addresses only. Taken together, this ensures a proper distinction of code and data, and prevents type confusion, for example, preventing the erroneous misinterpretation of an integer as an object reference. If instructions operate on types that cannot be determined statically, the verifier ensures that the bytecode contains the necessary casts and checks that are required to dynamically enforce type safety.
- Stack operations are balanced, i.e., there are no overflows or underflows of the operand stack.
- The class does not subclass a final class, and does not contain a method that overrides a final method.
- Every class except for `java.lang.Object` has a superclass.

Note that the above list presents important guarantees provided by the runtime's class file verifier, but this list is far from complete. In fact, a large portion of the Java Virtual Machine

Listing 2.9: Side-by-side comparison of C and Java with respect to memory handling

(a) Memory handling in C

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
 int n = atoi(argv[1]);
 int *v;

 // allocate memory for array
 v = (int*)malloc(n*sizeof(int));
 v[0] = v[1] = 1;

 for(int i = 2; i < n; i++) {
  v[i] = v[i-2] + v[i-1];
 }

 // output element at index 7
 printf("%d: %d\n", 7, v[7]);
 free(v); // deallocate memory
}
```

(b) Automatic memory handling in Java

```java
public class Main {

 public static void main(String[] a) {
  int n = Integer.parseInt(a[0]);
  // instantiate array
  int v[] = new int[n];

  v[0] = v[1] = 1;

  for(int i = 2; i < n; i++) {
   v[i] = v[i-2] + v[i-1];
  }

  String f = "%d: %d\n";
  // output element at index 7
  System.out.format(f, 7, v[7]);
 }

}
```

Specification is dedicated to class file verification, covering the topic at a level of detail that is not required for this thesis.

Statically verifying class files prior to their execution has several benefits. One major advantage is high performance. Verification is a one-time effort invested only once during class loading, but the guarantees provided by the verification process are valid throughout program execution. The amount of dynamic checks that need to be performed during program execution, e.g., to enforce type safety, are reduced to a minimum. Another advantage is the clean design of the solution. The JVM can be implemented in such a way that all code that is responsible for static class file verification is consolidated in a single dedicated component. The JVM's bytecode interpreter and JIT compiler can focus on their primary responsibility without having to implement a large amount of sanity checks.

2.2.3 Automatic memory management

The runtime implements various mechanisms to automatically take care of memory management. Specifically, this includes automatic allocation and deallocation of memory, as well as bounds checking. Assuming a bug-free implementation of the JVM and JCL, it should be impossible for a Java application to crash the JVM process by illegal memory access. This is a key difference to native code that typically requires application developers to manually take care of all memory-related concerns.

Listing 2.9 illustrates the difference between C and Java with respect to memory handling. Both code examples implement the same functionality. They dynamically create an array of

user-defined length and store Fibonacci numbers in it. Finally, they print the number that is stored at index seven of that array. Although functionally equivalent, there are language-specific differences between the two code examples. The C code needs to explicitly allocate memory for the array using `malloc()`, and it also needs to explicitly deallocate this memory using `free()`. The Java code simply instantiates and initializes the array using the `new` keyword, and the runtime automatically takes care of freeing the memory when no longer required. For this, the JVM's garbage collector runs iteratively to automatically free memory that is no longer referenced by the application.

Another important difference concerns array access. Both code examples print the contents of the array at a fixed index. Due to the fact that the array's length is user-defined, it may happen that this fixed index is out of the array's bounds. The effects of an out-of-bounds access in Java are well-defined: an `ArrayIndexOutOfBoundsException` will be thrown and the application can adequately respond to this or exit in a controlled way. In C, this illegal memory access results in undefined behavior. Depending on the compiler and operating system, the effects may differ and the underlying defect can potentially be hard to find. Application developers are thus strongly encouraged to prevent out of bounds access, e.g., by explicitly implementing thorough checks.[1]

Besides the memory-related features of the Java runtime highlighted above, there are also other important aspects of memory management that are automatically being taken care of by the Java runtime. Taken together, these mechanisms prevent by design that Java applications are susceptible to the same issues that typically affect native code, including stack and heap overflows, as well as memory leaks. Note, however, that the JVM itself is implemented natively, and thus potentially susceptible to the problems mentioned above.

[1] See "ARR30-C. Do not form or use out-of-bounds pointers or array subscripts" of the SEI CERT C Coding Standard for comparison

3

In-depth analysis of Java exploitation[1]

The Java platform has seen widespread adoption on various different platforms since its original release in the 1990s. However, its history is also a history of exploitation—all major versions of the runtime contained multiple severe security vulnerabilities that allowed attackers to fully bypass all security mechanisms. In this chapter, we will study in depth how attackers exploited the Java platform in the past and present, thus taking the best possible advantage of the multitude of attacks that are known to date. Our goal is to identify common attack vectors and other similarities among the different exploits in order to learn more about Java's weaknesses and shortcomings. The knowledge we gain here shall serve as a foundation for the development of suitable mitigation strategies.

3.1 Motivation and contributions

The Java platform has a history of about two decades, and the runtime has seen widespread distribution on desktop workstations, servers, and even consumer electronics. In terms of software, this is a long lifespan, but within this time, the platform has also been subject to a large number of attacks. In the severest of these attacks, adversaries bypass Java's security mechanisms in order to infiltrate the host machine that executes the runtime, thus fully compromising the target system.

As explained in the previous chapter, the Java runtime implements a complex security architecture with the goal of securely containing the execution of untrusted code in an isolated environment that provides only limited access to a well-defined set of system resources. This shall prevent, for example, that untrusted code can access sensitive information on the target machine without proper authorization, or maliciously depletes system resources to achieve a denial of service. For this, the security architecture combines a set of different mechanisms that need to work together in order to effectively provide the desired security guarantees, such as type safety.

Given the size and complexity of the Java runtime, and considering that the platform evolved over the course of decades, it comes as no surprise that all versions of Java SE had a large number of severe security vulnerabilities. Most of these defects allowed attackers to fully bypass the Java runtime's security mechanisms, thus achieving arbitrary code execution on the target machine.

[1] Parts of this chapter were taken directly or with modifications from [52].

As Cisco reports, Java was the top attack vector used by online criminals for web exploits in 2013 [20] and 2014 [21], with a share of at least 87%, thus leaving other web technologies far behind in that regard.

At the same time, with Java being deployed on billions of devices,[2] it is one of the most widely-used software systems. Java vendors like Oracle and IBM are thus avid to fix vulnerabilities in a timely manner. In recent years, the development of exploits and patches has turned into a continuous arms race between vendors and attackers. In some cases, however, security patches resembled rather cosmetic improvements that only blocked specific proof-of-concept exploits, but failed to address the underlying security defects that originally enabled the attacks. Consequently, it was possible to adapt certain blocked exploits with minor modifications such that they would work again on newer versions of the Java runtime that were long thought to be fixed.

One example is the exploit for CVE-2013-5838 that originally affected Oracle Java 7u25 and earlier. Due to an improper patch, it was possible to adapt the exploit with minimal changes so that it again affected Oracle Java 7u97 and Oracle Java 8u74 [100]. Another example is the exploit for CVE-2013-5456 that first targeted IBM SDK Version 7.0 SR5. Similar to the previous case, it was possible to modify the original exploit so that it again worked against IBM SDK Version 7.1 [99], a newer version of the platform that was thought to be fixed. Incidents like these lead to the impression that even for vendors it is hard to reason about Java's security architecture and properly maintain and evolve its security mechanisms over such long time periods. This once again emphasizes the need for a systematic analysis of the Java runtime's insecurities and shortcomings.

The primary goal of this chapter is thus to highlight the inner workings of past and current exploits to outline how the Java security architecture fails in practice. To this end, we conducted an in-depth study of 87 publicly available Java exploits found in the wild. We collected them, minimized them in order to reduce all unnecessary code and payloads, and integrated them into a custom test framework to facilitate systematic analysis. To the best of our knowledge, this is the largest sample set of Java exploits available to date. Based on their minimal representations, we conceptually split all reproducible exploits into independent, abstract steps that we call *primitives*. This allows us to compare and cluster different exploits on a conceptual level higher than the code, thus pointing us to weak spots of the Java platform which are exploited in many different attacks. Moreover, our findings indicate that the Java platform systematically lacks defense in depth, i.e., the security mechanisms it implements are not redundant, which allows that individual security defects can break the entire security architecture. This represents a design flaw that cannot simply be patched through minor changes.

In summary, we make the following contributions:

- a collection of 61 unique and reproducible Java exploits, based on an original set of 87 exploits,
- an analysis and categorization of these exploits in terms of behavior,
- an analysis of Java's security architecture in terms of weak spots, and
- potential security fixes for those weak spots.

We further make the full documentation of the exploit sample set publicly available.[3]

[2] `https://go.java/index.html`, accessed 27-Apr-2019

[3] `https://github.com/pholzinger/exploitstudy`

3.2 Creating an exploit sample set

The first step of this study was to create a sample set of real-world Java exploits found in the wild. Our aim was to collect a sufficiently large and diverse set of exploits as a basis for our analysis. To structure our efforts, we followed a multi-step process. First, we collected exploits from various online databases and exploits frameworks, including Metasploit[4] (22 exploits), Exploit-DB[5] (2), Packet Storm[6] (5), from the security research company Security Explorations[7] (52), and an online repository of Java exploits[8] (6). The result of this collection process was a total set of 87 original Java exploits. The majority of these exploits target the Oracle JDK (64), some the IBM JDK (22) and one is specific to Apple's JDK. Most exploits for Oracle's JDK can also be run on other vendors' JDKs, as they involve security vulnerabilities in the very core of Java which is implemented similarly by the different vendors. The associated CVE identifiers, where available, range from 2003 to 2013. We tagged all original exploits with a unique identifier to allow for easy tracking throughout the analysis.

After our collection process, we thoroughly tested all samples to verify that the exploits were actually effective. To do so, we created a common testing framework that automatically sets up a testing environment and executes the exploit code. For exploits that attempt to disable the security manager to achieve arbitrary code execution, our framework puts the default security manager in charge of policy enforcement, runs the exploit and subsequently checks whether the security manager was set to `null`. This allows for testing all such exploits in a uniform and fully-automated manner. For exploits that do not aim for disabling the security manager we tested the effectiveness manually and used our testing framework merely to automate exploit execution. This includes all denial-of-service and information-disclosure attacks. We removed from any further consideration all exploits that we were unable to run successfully.

Most exploits for the Oracle JDK that we were able to test successfully run on Java 1.6 or Java 1.7, few exploits require Java 1.4 or 1.5. All exploits for the IBM JDK that we were able to reproduce successfully run on IBM JDK 7.0-0.0 or 7.0-3.0.

Some of the downloaded exploits were hard to review manually because they contained large byte arrays with possible payloads. Some of them also contained graphical user interface components, unnecessary reflection constructs, and in some instances even bugs. As a next, we thus transformed all exploits that we tested successfully into *minimal exploits*. They contain only those lines of code that are crucial for the exploit to work. Also, as far as possible, we replaced the large byte arrays with the code they contained, creating the byte arrays only on demand to facilitate manual code reviews. In some cases, however, this was not possible because the very nature of the exploit was to work with bytecode that cannot be produced from source.

Finally, we compared all sources that we acquired manually and merged those that were semantically equivalent. At the end of this process, we ended up with 61 unique and reproducible

[4] `https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits`, accessed 09-Nov-2018

[5] `https://www.exploit-db.com`, accessed 09-Nov-2018

[6] `https://packetstormsecurity.com`, accessed 09-Nov-2018

[7] `http://www.security-explorations.com/en/SE-2012-01-poc.html`, accessed 09-Nov-2018

[8] `https://bitbucket.org/bhermann/java-exploit-library`, accessed 09-Nov-2018

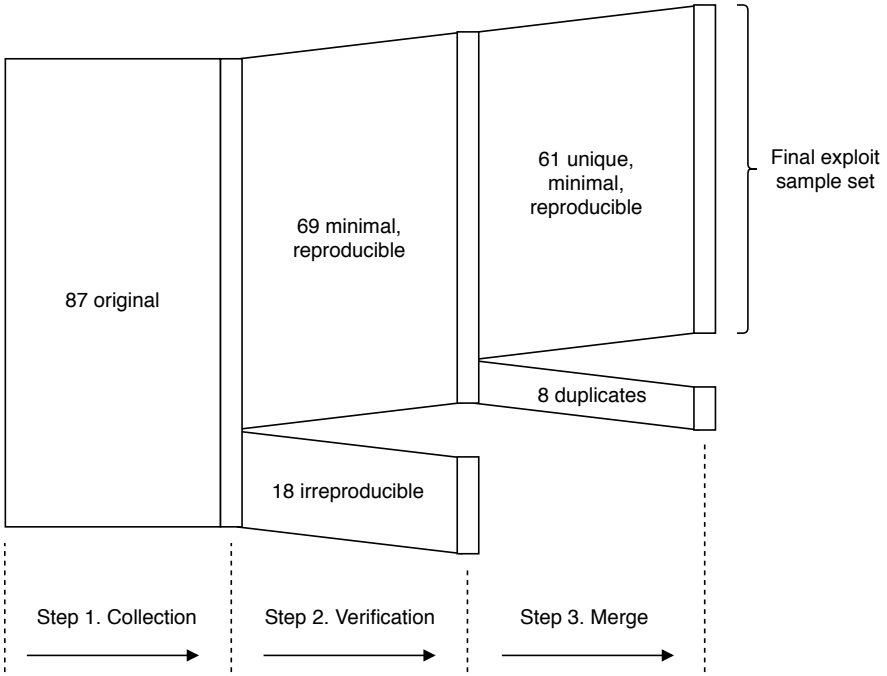


Figure 3.1: Overview of the workflow for creating the exploit sample set

exploits that we used as a basis for the analysis. Figure 3.1 visualizes this workflow and quantifies the exploits that were removed from the set at each stage of the process.

3.3 Modeling exploit behavior

3.3.1 Exploit behavior

The goal of this work is to understand how attackers exploit the Java platform, and to identify measures of improvement by analyzing the behavior of a large body of exploit samples. The first essential question that needs to be discussed is the definition of *behavior* that will be used in the context of this analysis.

Instead of providing an abstract, formal definition of the term, let us consider Listings 3.1 and 3.2. Each of the two listings contains a method, `getClass1()` and `getClass2()`, which is able to dynamically load a class. Since both methods make use of security vulnerabilities for this, an exploit that has not been granted any privileges can use them to load arbitrary classes, including restricted ones. As previously explained in Section 2.2.1, this poses a security risk, as restricted classes may provide functionality that can be used to disable security checks. While `getClass1()` and `getClass2()` implement the exact same functionality, they use different implementations to achieve their goal; `getClass1()` uses classes `JmxMBeanServer` and `MBeanInstantiator`, and `getClass2()` depends on `MethodHandles` instead. If some developer were to document the behavior of any of these methods, one intuitive way would be to add

Listing 3.1: Modified excerpt of an exploit for CVE-2013-0431

```
1 // Method loads arbitrary classes
2 private Class getClass1(String s) {
3   JmxMBeanServer server=(JmxMBeanServer)JmxMBeanServer.
        newMBeanServer("",null,null,true);
4   MBeanInstantiator i=server.getMBeanInstantiator();
5   return i.findClass(s,(ClassLoader)null);
6 }
```

Listing 3.2: Modified excerpt of an exploit for CVE-2012-5088

```
1 // Method loads arbitrary classes
2 private Class getClass2(String s) {
3   MethodType mt=MethodType.methodType(Class.class,String.class);
4   MethodHandles.Lookup l=MethodHandles.publicLookup();
5   MethodHandle mh=l.findStatic(Class.class,"forName" mt);
6   return (Class)mh.invokeWithArguments(new Object[]{s});
7 }
```

a code comment similar to the one in line 1 of Listing 3.1 and 3.2, respectively. It simply states that they can be used to *load arbitrary classes*. This is on the right level of abstraction for another developer to understand the purpose the methods, that they implement the same functionality and that they can thus be used interchangeably. This is also the right level of abstraction for describing the behavior of exploits in the sample set, such that it allows for identifying common attack patterns and frequently abused weak points in the Java platform. For instance, if the analysis revealed that every single exploit in the entire sample set uses vulnerabilities to dynamically load arbitrary classes, this could be seen as a clear indication that the measures implemented to prevent the loading of restricted classes by untrusted code are fragile and insufficient. This is the kind of evidence-based conclusion this exploit analysis is aiming for. Details about how the exploits implement this functionality are not required to draw this conclusion. However, these implementation details can help in understanding why existing countermeasures fail and may influence the development of new countermeasures.

The behavior of an entire exploit that disables all security checks is more complex than the code examples provided in the listings. Functionality to load arbitrary classes could be one building block of such an exploit, but a complete description of a full-bypass exploit may require more than one building block to adequately model its behavior on this level of abstraction.

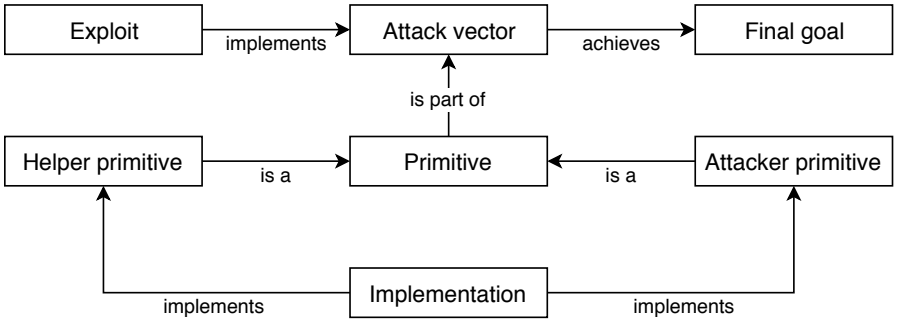


Figure 3.2: Meta model used to document exploits

3.3.2 A meta model to document exploits

For purposes of this exploit analysis, we developed a new meta model that we used as a basis for documenting the exploits in the sample set. Creating this model was guided by the following requirements.

- The meta model should focus on behavior (as defined informally in Section 3.3.1) and abstract from implementation, i.e., specific bug details. Only with this layer of abstraction it is possible to identify commonalities between the different exploits, as many of them use entirely different implementations.
- Our definition of behavior is at a rather low level of abstraction. The model must thus allow for documenting behavior in terms of reusable building blocks, which can be combined to model complete attacks.
- The analysis shall not only focus on how exploits abuse vulnerabilities, but also on how they make use of specific features of the Java platform, such as reflection or method handles.

Guided by the above requirements, we developed a new meta model that we instantiated to document all exploits in the sample set. Note that this meta model is by no means specific to Java exploits. In fact, it is a general model that can be used to document the behavior of malicious code in entirely different attack domains. The following introduces the seven basic entities of this model and Figure 3.2 provides an overview.

Final goal

This is the most abstract entity in the meta model. It is used to describe the final goal of an entire attack by means of a brief textual description. Final goals express the way in which an attack vector is considered to be malicious. One final goal can be achieved through one or more attack vectors, but each attack vector achieves only one final goal. More than one final goal may be needed to document an entire set of exploits, as different exploits may implement different attack vectors.

Examples: Information disclosure, denial of service

Attack vector

An attack vector is one way to achieve a specific final goal. It is composed of one or more primitives. Two attack vectors are similar, if they are composed of the same set of primitives.

Primitive

A primitive is a building block of a vector that describes specific behavior. Primitives are more abstract than implementations, but less abstract than vectors. All primitives describe behavior at the same level of abstraction. Each primitive can be used as a building block for more than one vector. There are two kinds of primitives: helper primitives and attacker primitives. All primitives are documented by a set of properties, including a title, a unique identifier, a textual description, preconditions, etc. Each primitive is instantiated by at least one implementation.

Attacker primitive

An attacker primitive is one specific kind of primitive that describes the exploitation of a security vulnerability. It violates the security model of the target system, which must be properly documented in its description. Each attack vector must be composed of at least one attacker primitive.

Example: Load arbitrary classes

Helper primitive

A helper primitive is, besides attacker primitives, another specific kind of primitive. It describes intentionally introduced features of the Java runtime, as opposed to attacker primitives, which resemble security vulnerabilities introduced unintentionally. Each helper primitive is a counterpart to at least one attacker primitive, in the sense that the corresponding attacker primitives would be useless, or at least less useful without the helper primitive. A helper primitive's description must explain how it adds value to attacker primitives. Helper primitives are optional elements of attack vectors, as not all attacker primitives rely on helper primitives.

Example: Set of restricted classes that set a specified field accessible

Implementation

Implementations are specific code sequences or APIs that instantiate primitives. As such, they are at the lowest level of abstraction in the meta model. Each implementation instantiates only one primitive, but a primitive can have multiple implementations.

Example: The codes in Listing 3.1 and Listing 3.2 are two implementations for the primitive "load arbitrary classes".

Exploit

An exploit is a concrete instance of an attack vector. It represents an executable combination of implementations for the specific primitives of the attack vector. Every exploit implements a single attack vector, but two or more exploits can implement the same attack vector, using

different implementations. As an example, one exploit makes use of the code of Listing 3.1, and another exploit uses the code of Listing 3.2 instead. If this is the only difference between those two exploits, we consider them to be different exploits that implement the same vector.

3.3.3 Documenting the exploit sample set

The basis of our analysis is the set of minimal exploits that we integrated into our common testing framework. Each minimal exploit is based on at least one original exploit that we obtained online, with all unnecessary code removed. All minimal exploits are different in the sense that they either implement different vectors, or they implement the same vectors using different vulnerabilities or features of the platform.

Documenting the behavior of the exploits in this sample set requires us to instantiate the meta model that we presented in Section 3.3.2. This means, we have to specify a set of final goals, primitives, attack vectors, etc. that closely resemble the behavior of the actual exploits. The meta model we developed merely describes how to structure a behavioral description of exploits, it does not provide any guidance or a process that needs to be followed in order to instantiate the model based upon source code. For this, as we elaborate in the following, we chose an iterative approach with redundant supervision.

The first step of this effort is to identify final goals. This is a reasonable way to start the documentation process, as final goals describe exploit behavior at the highest level of abstraction and their identification requires little knowledge about implementation details. After reviewing the entire sample set, we found that a variation of the classic CIA triad [88] appropriately reflects the attack goals:

- **Information disclosure** (3 exploits)
  There are exploits in the sample set that reveal sensitive information about the target system, thus violating confidentiality.
- **Full bypass** (56 exploits)
  By far the largest portion of exploits in the sample set aims for arbitrary code execution on the target machine, thus fully compromising the host that executes the Java runtime.
- **Denial of service** (2 exploits)
  Few exploits attack the availability of the target system, without achieving information disclosure or arbitrary code execution.

The second step in describing exploit behavior is to document for each exploit the set of primitives it uses to achieve its final goal. For this, we thoroughly inspected all exploits in detail to understand which vulnerabilities and runtime features they use to perform the attacks. This step required multiple iterations to ensure that all primitives we describe are at the same level of abstraction. Naturally, there is a certain design space when choosing appropriate primitives to model exploit behavior—they are not given, and there is no ground truth. However, the specification of primitives was not done arbitrarily, but, as we explain in the following, supported by guidance.

The specification of new attacker primitives was triggered by the security vulnerabilities the exploits use. Each security vulnerability, by definition, violates the security model. Different vulnerabilities may violate the model in the same way or in different ways; they could depend

Table 3.1: Helper primitives

ID	Title
H1.	Load arbitrary classes if caller is privileged
H2.	Lookup MethodHandle
H3.	Get access to declared methods of a class if caller is privileged
H4.	Get access to declared field of a class if caller is privileged
H5.	Get access to declared constructors of a class if caller is privileged
H6.	Set of restricted classes that define a user-provided class in a privileged context
H7.	Set of restricted classes that set a specified field accessible
H8.	Set of restricted classes that provide access to declared fields of non-restricted classes
H9.	Use confused deputy to lookup method handle
H10.	Private `PrivilegedAction` that provides access to arbitrary no-argument methods and sets them accessible

on different prerequisites or cause different postconditions. All those characteristics are part of a primitive's description. For each vulnerability, we evaluated whether there is an already existing attacker primitive with a matching description. If this was not the case, we either specified an entirely new primitive, or we adapted the closest match in the set of already existing primitives.

The specification of new helper primitives was guided differently. As opposed to attacker primitives, those are not associated with vulnerabilities, i.e., unintended behavior, but rather with intended behavior, i.e., features of the runtime. The Java platform is feature-rich, and implementing even just simple applications requires heave use of the JCL. However, not all parts of the class library used by exploits are of equal interest from a security point-of-view. We hence limited our view to three specific features of the platform: dynamic class loading, reflection, and method handles. Due to their very nature, we assume that those parts of the class library pose a risk to the proper implementation of the Java security model. Documenting how exploits make use of these features may point to weaknesses in the Java security architecture that otherwise could easily be overlooked.

At any stage of developing the model we applied redundant supervision: the specification of final goals and primitives, and the documentation of all exploits has been assessed by three analysts. Any misunderstandings or disagreements were resolved in group discussions. The result of our documentation efforts is a set of three final goals, 27 attacker primitives, and ten helper primitives. Each exploit is associated with an attack vector, composed of one or more primitives, each of which is instantiated by one implementation. This documentation is the basis for the analysis and conclusions in the subsequent sections.

Table 3.1 provides an overview of all helper primitives used for exploit documentation, and Table 3.2 shows the corresponding set of attacker primitives. We further make the full

Table 3.2: Attacker primitives

ID	Title
A1.	Access to system properties
A2.	Load arbitrary classes
A3.	Load restricted class
A4.	Call arbitrary public methods
A5.	Access to arbitrary public method
A6.	Access to MethodHandles for arbitrary protected methods
A7.	Use system class to call arbitrary MethodHandles
A8.	Get access to declared method of a class
A9.	Get access to declared field of a class and set it accessible
A10.	Get access to declared, non-static fields of a serializable class and set them accessible
A11.	Read and write value of an arbitrary non-static field
A12.	Get access to declared method of a class and set it accessible
A13.	Get access to public constructors of a class
A14.	Define class in a privileged context
A15.	Set arbitrary members accessible
A16.	Restricted field manipulation
A17.	Use system class to call arbitrary static methods
A18.	Call arbitrary method in privileged context
A19.	Call arbitrary instance method in privileged context
A20.	Use system class to call arbitrary methods
A21.	Use a system class to call a subset of methods
A22.	Instantiate arbitrary objects
A23.	Instantiate a subset of restricted classes
A24.	Create very large file
A25.	Call arbitrary method in trusted method chain
A26.	Access to MethodHandle of constructor of private inner class
A27.	Unlimited nesting of Object arrays

documentation of the exploit sample set publicly available, which also includes a more extensive documentation of all primitives.[9]

3.4 Analysis and findings

In the following we use the extensive documentation of the 61 minimal exploits to provide insight into how attackers use specific vulnerabilities and features of the Java platform to implement their attacks. Due to the complexities involved in exploit implementations, we cannot provide a detailed view on the exploits' behavior on the level of primitives or implementations, as this would clearly exceed any space restrictions. Instead, we derived a smaller set of higher-level weaknesses from the primitives that we used to document the exploits, as well as their implementation details. Based on this, we will discuss the following research questions.

RQ1: What are the weaknesses attackers exploit to implement their attacks?
RQ2: How do attackers combine the weaknesses to attack vectors?

While RQ1 discusses the weaknesses that exploits abuse in isolation, RQ2 is dedicated to an analysis of how attack vectors combine multiple weaknesses.

3.4.1 Commonly exploited weaknesses

To address RQ1, we derived a set of nine higher-level weaknesses from the full documentation of the 61 minimal exploits that are commonly utilized by the exploits in the sample set. These weaknesses are well-suited for providing an overview as they combine multiple related primitives and implementations. Each weakness represents a specific kind of vulnerability or runtime feature used by at least 10% of all exploits, as can also be seen in Table 3.3. Note that some primitives are associated with more than one weakness, and that one exploit can make use of more than one weakness. The following explains in detail how attackers utilize the different weaknesses in their exploits.

W1: Caller sensitivity

Related primitives: H1, H3, H4, H5

The JCL contains a large number of so-called *caller-sensitive methods*. These methods vary their behavior depending on their immediate caller, e.g., skip permission checks if only the immediate caller is trusted. For this, caller-sensitive methods request information about their caller by invoking `sun.reflect.Reflection.getCallerClass()` or similar methods. With the implementation of JEP 176 [94] in Java 8 all caller-sensitive methods were marked with the `@CallerSensitive` annotation, which makes such methods easy to find. Before this change, there was no trivial way to distinguish caller-sensitive methods from caller-insensitive methods.

In our sample set, caller-sensitive methods are abused by 22 minimal exploits for the following purposes.

[9] `https://github.com/pholzinger/exploitstudy`

Table 3.3: Weaknesses commonly utilized by exploits in the sample set

Weakness	# exploits
Unauthorized use of restricted classes (W5)	32 (52%)
Loading of arbitrary classes (W4)	31 (51%)
Unauthorized definition of privileged classes (W6)	31 (51%)
Reflective access to methods and fields (W8)	28 (45%)
Confused deputies (W2)	22 (36%)
Caller sensitivity (W1)	22 (36%)
Method handles (W9)	21 (34%)
Serialization and type confusion (W7)	9 (15%)
Privileged code execution (W3)	7 (11%)

- All 22 exploits use caller-sensitive methods, primarily `Class.forName()`, to load arbitrary classes.
- Out of those 22 exploits, 13 use caller-sensitive methods to bypass Java's rules for information hiding, i.e., get reflective access to members of classes they should not have access to.

Caller-sensitive methods are not vulnerabilities by themselves, as their behavior is intended. They can only be abused by malicious code if called through a confused deputy. Because of this, we modeled all caller-sensitive behavior abused by exploits as helper primitives. Even though the actual vulnerabilities are the confused deputies, caller-sensitive methods significantly increase the attack surface—without these methods, many confused-deputies that do not explicitly elevate privileges would not have to be considered security vulnerabilities.

In addition to the fact that caller-sensitivity increases the attack surface, we also consider the entire concept of caller-sensitivity as counter-intuitive when applied to security checks. After all, it grants privileges to callers implicitly, without those callers being aware of those privileges.

W2: Confused deputies

Related primitives: A7, A17, A20, A21

The confused deputy problem was already well-known as a concept in the area of computer security before the introduction of Java [48]. In general terms it refers to a trusted entity, for example an application, that can be instructed by attackers to abuse its legitimately granted rights to fulfill the attacker's goal.

In the context of Java security, confused deputies are higher-privileged classes that can be abused by lower-privileged classes, e.g., to call methods that would otherwise be inaccessible. Moreover, many confused deputies that are part of the JCL can be abused by attackers to invoke caller-sensitive methods. Calling a method through a confused deputy will not allow for bypassing arbitrary permission checks, as it will not elevate privileges. However, some caller-sensitive methods skip permission checks if the immediate caller is trusted. Thus, calling certain methods through a system class can be profitable to an attacker.

Listing 3.3: Simplified example code to illustrate a confused-deputy vulnerability

```
1 Class SystemClass {
2   public Object invoke(Method m, Object[] args) {
3     return m.invoke(this, args);
4   }
5 }
```

Out of the 61 minimal codes, 22 exploits make use of confused deputies. As we elaborate in the following, the underlying vulnerabilities are caused by different issues. Note that some exploits make use of more than only one confused deputy.

- Ten exploits abuse a confused deputy that allows for calling arbitrary static methods. In nine cases, the vulnerability was caused by a trusted class implementing a method similar to the code in Listing 3.3. In this example, method `SystemClass.invoke()` receives a `Method` object by the caller, as well as call arguments, and then invokes that method using `Method.invoke()`. The first argument to `Method.invoke()` is the instance upon which to perform the call. In this example, it is always `this`, i.e., an instance of class `SystemClass`. The second argument is an array of arguments. Just by reviewing this method, it seems impossible for any caller to use `SystemClass.invoke()` to invoke a method outside of `SystemClass` as the first argument to `Method.invoke()` is always `this`, pointing to an instance of `SystemClass`. However, this is only true for instance methods, but not for static methods. If `Method.invoke()` is called on a static method, the first argument will be ignored. Attackers can thus use a class like `SystemClass` as a confused deputy to call arbitrary static methods, including those of restricted classes.
  We should consider the implementation of `Method.invoke()` as the actual root cause of these vulnerabilities, as ignoring arguments is counter-intuitive and bad style. There are various ways on how to implement this such as to avoid usability issues. One simple way would be to modify `Method.invoke()` such that it checks if the instance of `Method` represents a static method. If that is the case, it should further check the first argument supplied to `invoke()`, and rather than ignoring it as the current implementation does, it should throw an `IllegalArgumentException` or an `IllegalAccessException` if it is not `null`. This implementation would immediately render impossible attacks on classes similar to `SystemClass` in the example in Figure 3.3, while leaving the method signature of `Method.invoke()` unchanged.
- Four exploits abuse a defect in the implementation of method handles. An example for this is presented in Listing 3.2. Untrusted code can use `MethodHandle.invokeWithArguments()` as a wrapper to `MethodHandle.invokeExact()`, which will then call the target method. The problem with this is that caller-sensitive methods invoked this way will incorrectly determine `MethodHandles.invokeWithArguments()` as the immediate caller, instead of the untrusted code that actually called `invokeWithArguments()`. Since `invokeWithArguments()` is declared in a trusted class, many caller-sensitive methods will skip a permission check and thus expose sensitive functionality to malicious code. This

problem illustrates how error-prone caller-sensitive behavior is in practice. Determining the immediate caller is by no means a trivial lookup on the call stack. The Java runtime has to skip certain methods of the reflection API and method handles on the call stack to ensure that caller-sensitive methods called this way behave exactly as they would if called immediately.
- Ten exploits abuse confused deputies that were introduced by various complex vulnerabilities, which only allow for calling an implementation-specific subset of methods. Due to the diversity of these vulnerabilities there is no common root cause.

W3: Privileged code execution

Related primitives: A18, A19, A25

We differentiate between privileged code execution and confused deputies. The confused deputies we referred to in the previous paragraphs allowed untrusted code to route a call sequence thorough a system class, such that the immediate caller of the actual target method would be the trusted system class, and not the malicious code that triggered the call sequence. This allows an attacker to profit from caller-sensitive methods. In contrast, privileged code execution refers to vulnerabilities that allow an attacker to execute code in a way that it successfully passes arbitrary permission checks. This is thus more powerful than the confused-deputy vulnerabilities we described before, as they are not dependent on caller sensitivity.

There are two different ways how exploits achieve privileged code execution:
- Four exploits abuse system classes that explicitly elevate privileges and then call attacker-provided methods with arbitrary arguments. These vulnerabilities are specific to the IBM Java platform. Since privilege elevation is done explicitly, and the implementation of these vulnerabilities is rather simple, we assume that static analysis can be used to find instances of this problem.
- Three exploits make use of more complex vulnerabilities to achieve what is known as trusted method chaining [58]. In trusted method chaining, malicious code is able to setup a thread with only trusted system classes that will eventually execute a call sequence that is profitable to the attacker. This is possible through, e.g., attacker-provided scripts that will be evaluated dynamically by a trusted class. Because the entire call stack of the running thread only contains trusted system classes, all permission checks will succeed. A simple proposal to address this issue systematically is adding the class that initiates a thread to the beginning of the newly created thread's call stack. Whether this is feasible without any unwanted side effects needs to be properly evaluated.

As can be seen from the numbers above, cases of explicit privilege escalation are rare. While there are only four exploits in the sample set that abuse vulnerabilities of this kind, there are more than 20 exploits that abuse confused deputies caused by the implicit elevation of privilege. This indicates that explicit privilege elevation is easier to control than implicit privilege elevation.

W4: Loading of arbitrary classes

Related primitives: H1, A2, A3, A22, A23

Dynamic class loading is a central security-related feature of the Java platform. Class loaders in the JCL are supposed to ensure that all code is only able to load classes that it is allowed to

access. Yet, we find that 31 out of 61 minimal exploits are able to load classes they should be incapable of loading.

- Most commonly (20 exploits), malicious code abuses a system class as a confused deputy to invoke a caller-sensitive method, e.g., `Class.forName(String)`, which will use the immediate caller's defining class loader to load the requested class. Since in this setting the immediate caller of `forName()` is a trusted system class, and its defining class loader is the bootstrap class loader, untrusted code can request the loading of arbitrary restricted classes. Listing 3.2 gives an example for this. We modeled the various confused-deputy defects as instances of attacker primitives, and the corresponding caller-sensitive methods as helper primitives.
- The remaining eleven exploits abuse other security vulnerabilities to load or instantiate classes that should be inaccessible to them. We reviewed those vulnerabilities and found that the underlying defects are rather diverse. An example for this is provided in Listing 3.1. In this example, the vulnerability is in a trusted class, `MBeanInstantiator`, which simply provides an unrestricted public interface for loading arbitrary classes. In another case, a complex call sequence will allow untrusted code to define a custom class using a special class loader. This special class loader will not define attacker-provided classes within privileged protection domains, but the class loader itself has the capability to load arbitrary classes. A custom class that as been defined with this loader can thus simply call `Class.forName()` which will use the caller's defining class loader to load arbitrary classes.

The evaluation of these 31 exploits highlights confused-deputy defects in combination with caller sensitivity as a major issue. There is no inherent reason for why public interfaces for class loading should be caller sensitive. While immediately removing these methods and replacing them by caller-insensitive counterparts would break backward compatibility, one should consider their deprecation. The remaining vulnerabilities that allow for arbitrary class loading are too diverse to be addressed by a single solution. To fix them, a major redesign of the class loading mechanism would be required.

W5: Unauthorized use of restricted classes

Related primitives: H6, H7, H8, A23

Access to restricted classes greatly contributes to the insecurity of the Java platform. In total, 32 out of 61 minimal exploits make immediate use of at least one restricted class. Exploits in the sample set use them for one or more of the following purposes:

- Defining a custom class in a privileged context (used by 22 exploits). This is highly valuable to an attacker, as it allows for arbitrary code execution. A custom class defined in this way can disable the security manager without having to bypass any further security checks.
- Accessing members of non-restricted classes (used by three exploits). Access to private or protected methods of system classes violates information hiding and exposes sensitive functionality to untrusted code.

- Setting specific fields accessible (used by nine exploits). There are certain vulnerabilities that will provide access to declared members of a class. However, for untrusted code to be able to use private fields and methods obtained this way, they must first be set accessible.
- One exploit is able to instantiate a subset of restricted classes that can be used for information disclosure.

Note that this does not even include the uses of `sun.awt.SunToolkit`, which we treated differently than all the other restricted classes. While we generally consider primitives that involve the use of a restricted class as helper primitives, we consider primitives that involve `SunToolkit` as attacker primitives. The difference is that restricted classes other than `SunToolkit` cannot be accessed by untrusted code without exploiting a security vulnerability, whereas it was always possible to access `SunToolkit` without violating the security model: there is a publicly accessible field of type `java.awt.Toolkit` which contains an instance of a platform-specific toolkit that in turn extends `sun.awt.SunToolkit`. Using `Class.getSuperClass()` then provides access to `SunToolkit`.

As explained in Section 2.2.1, security-sensitive functionality in restricted classes is not protected by Java's general approach to access control by means of stack inspection. Instead, restricted classes are protected in a capability-based manner, i.e., whenever untrusted code gets a hold of an instance of a restricted class, it can use it without having to bypass any further checks. The heavy use of restricted classes in the exploit sample set illustrates that this entire concept is very hard to implement securely.

We further found that the number of restricted packages increased significantly over time, which is a dangerous trend. Java 1.7.0 contained four restricted packages (not counting subpackages), version 1.7.0u11 contained eight, and Java 1.8.0u92 already contained 47 restricted packages. Even though the case of `SunToolkit` is exceptional, it once again demonstrates how hard it is to protect all instances of restricted classes from being leaked to untrusted code. This is clearly a major design issue that complicates maintenance of the Java platform and weakens its security guarantees in practice. Ideally, the concept of restricted classes and the capability-based way of protecting them would be dropped in favor of proper permission checks.

W6: Unauthorized definition of privileged classes

Related primitives: H6, A6, A14

Defining a class in a protection domain that is associated with all permissions allows for arbitrary code execution. This is achieved by 31 out of 61 minimal exploits, using one of the following three ways:

- 22 exploits use a set of restricted classes to define a custom class with all privileges. This obviously requires an attack vector that abuses vulnerabilities to get access to methods in restricted classes in the first place. Restricted classes should be changed such that they only define privileged classes if absolutely needed. Further, such sensitive methods should be guarded by proper permission checks. It may be possible to implement these changes even without breaking backward compatibility, however, this requires further investigation.
- Two exploits obtain unauthorized access to method handles for arbitrary protected methods. This can be used to call internal methods of class loaders immediately, thus bypassing any security checks implemented in publicly accessible methods.

- Seven exploits abuse other, more complex vulnerabilities to immediately define custom classes with all permissions.

W7: Serialization issues and type confusion

Related primitives: A3, A11, A14, A16

Nine minimal exploits make use of either serialization issues, type confusion, or a combination of the two. As we explain in the following, the effects of using such vulnerabilities can be very different.

- Two exploits use a deserialization sequence to instantiate a custom class loader, which can be used to define a class with higher privileges.
- One exploit uses deserialization within a custom thread, to have a restricted class be loaded by the bootstrap class loader.
- Two exploits use serialization issues to bypass information hiding, but in different ways. One of the two exploits, involving CVE-2013-1489, requires a preparation step before the actual attack can be carried out. For this, the attacker manipulates an instance of a system class in a way that would be impossible when running with limited privileges. Specifically, this concerns the manipulation of the value of a certain private field of that system class, which holds a bytecode representation of a class. When carrying out the actual attack, the exploit merely contains a deserialized version of this manipulated instance of a system class. The malicious code then deserializes this manipulated instance on the target machine, and triggers a call sequence that will cause the manipulated instance to define the malicious bytecode in a domain that provides access to restricted classes.
  The second exploit that uses serialization to bypass information hiding uses a custom output stream to leak declared fields of serializable classes, while their instances are about to be written. This allows for manipulating private fields of system classes.
- Two exploits use type-confusion vulnerabilities to confuse a system class with a spoofed class, e.g., `AccessControlContext` and `FakeAccessControlContext`, to bypass information hiding. The spoofed class declares similar fields as the system class, but it uses `public` modifiers for fields that are declared as private fields in the system class. Due to the type confusion, the runtime will allow untrusted code to access fields that are actually private.
- Two exploits combine serialization and type confusion to implement an attack. One of them uses serialization for similar purposes as the exploit involving CVE-2013-1489. As explained above, it modifies private fields of a system class before the actual attack and then only deploys the serialized object, which can be deserialized by untrusted code at any time, even though its running with limited privileges. Next, it uses this system class to confuse a spoofed class loader with the application class loader in order to be able to define a privileged class.
  The other exploit uses a custom input stream to perform type confusion during deserialization. As already explained above, it also uses this to confuse a spoofed class with a system class, which both declare the same fields, but with different visibility modifiers. By this, the exploit gets access to private fields of system classes.

Listing 3.4: Vulnerability in `sun.awt.SunToolkit` in Java 7

```
1  public static Field getField(final Class klass, final String
       fieldName) {
2    return AccessController.doPrivileged(new PrivilegedAction<Field
         >() {
3      public Field run() {
4        try {
5          Field field = klass.getDeclaredField(fieldName);
6          assert (field != null);
7          field.setAccessible(true);
8          return field;
9        } catch (SecurityException e) {
10         assert false;
11       } catch (NoSuchFieldException e) {
12         assert false;
13       }
14       return null;
15     }//run
16   });
17 }
```

W8: Reflective access to methods and fields

Related primitives: H3, H4, H5, H7, H8, H10, A5, A8, A9, A10, A12, A13, A15
Improper uses of reflection in system classes, and certain caller-sensitive methods can be used by malicious code to bypass information hiding. In total, 28 minimal exploits achieve this by abusing various different vulnerabilities and helpers.

- 16 exploits use a vulnerability that will not only provide untrusted code access to declared fields or methods of a class, i.e, provide instances of `Field` and `Method`, but also set these instances accessible, which allows for the invocation of even private methods, and unrestricted read and write operations to fields. To achieve this, exploits frequently use functionality implemented in `sun.awt.SunToolkit`. Listing 3.4 shows how `SunToolkit.getField()` provides unrestricted access to declared fields, provided that the caller has access to the `Class` instance that represents the target type that declares the desired field.
- 13 exploits use confused deputies to invoke caller-sensitive methods, such as `Class.getDeclaredFields()` and `Class.getDeclaredMethods()`.
- Twelve exploits make use of other issues to access members.

The fact that so many exploits make use of reflection to break information hiding clearly shows that a reflection API is hard to implement securely. We cannot present a solution to the manifold issues without a significant redesign that would break backward compatibility.

W9: Method handles

Related primitives: H2, H9, A6, A26

Similar to the reflection API, method handles can be used to bypass information hiding. While there are certain commonalities, there are also interesting differences, as we show in the following.

- Twelve exploits abuse a confused deputy to call `MethodHandles.lookup()` to get a lookup object on behalf of a system class. Such a lookup object can be used by malicious code to access members that are accessible to the system class, but that should be inaccessible to untrusted code. Malicious code does not have to bypass any security checks to get access to class members once it gets a hold of this lookup object—similar to restricted classes, the runtime tries to prevent that untrusted code gets access to such powerful lookup objects in the first place.
- Without using any security vulnerabilities, eight exploits make regular use of `MethodHandles.lookup()`, or `MethodHandles.publicLookup()` to access members. In most cases, this is simply done because other vulnerabilities depend on method handles, as illustrated in Listing 3.2. In other cases, however, method handles have been deliberately used as an alternative to the reflection API, because they can be less strict with respect to type checking. This is important for a few rare cases of type confusion. During testing, we found that using the reflection API to access members of a confused type resulted in an exception due to a type mismatch, while using method handles worked without any errors. While this flexibility of method handles is advertised as a feature, it is also helpful to attackers
- Three exploits use other vulnerabilities that provide untrusted code access to method handles that should be inaccessible.

3.4.2 Combinations of weaknesses in attack vectors

An attack vector consists of one or more primitives, and may hence combine multiple weaknesses to form one attack. The entire set of 61 minimal exploits implements 33 different attack vectors. The total number of vectors is smaller than the total number of exploits, because two exploits can implement the same vector, i.e., the same set of primitives, but using different implementations.

To address RQ2, we evaluated how exploits combine the different primitives to attack vectors and found that there are three different categories of attacks. As illustrated in the attack tree in Figure 3.3, these categories are *single-step attacks*, *restricted class attacks*, and *multi-step attacks*. The following describes each category in detail.

Single-step attacks

The category of single-step attacks comprises 13 of the 33 vectors, implemented by 28 minimal exploits. These vectors have in common that they are of length one and comprise only a single attacker primitive that can be used alone to achieve the final goal. In one exceptional case the exploit uses an additional helper primitive. All five exploits that achieve denial of service or information disclosure belong to this category, as well as the seven exploits that achieve privileged code execution. Another seven exploits use security vulnerabilities to immediately define a custom class with higher privileges, thus achieving full bypass without relying on any

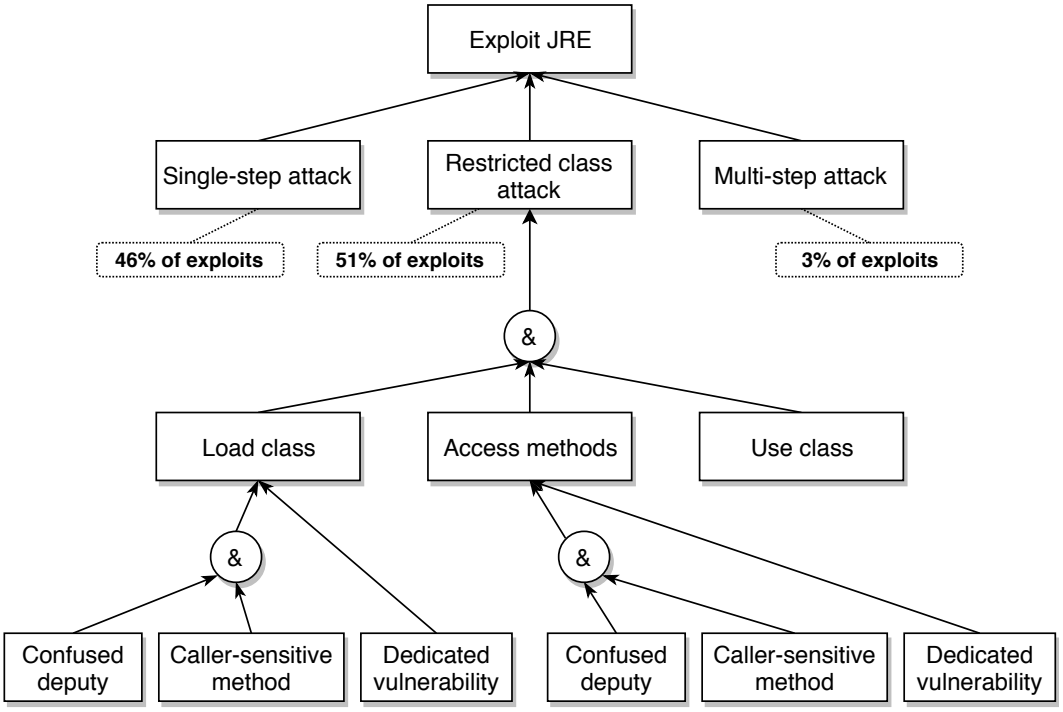


Figure 3.3: Three common attack vectors implemented by exploits

other vulnerabilities. Six exploits perform unauthorized manipulation of field values, which can be used alone for privilege escalation. The remaining two exploits use vulnerabilities to get access to method handles that reference arbitrary protected methods. These single-step attacks are hard to mitigate systematically, as they exploit individual vulnerabilities of various types in different components.

Restricted class attacks

The category of restricted class attacks comprises 18 vectors, implemented by 31 exploits. They all make immediate use of a restricted class and combine multiple primitives to achieve a final goal. As illustrated in Figure 3.3, most of them comprise three common steps: (a) load restricted class, (b) get access to methods of that restricted class, (c) use the restricted class by calling its methods. We found that eleven of those 18 vectors, implemented by 22 exploits, use a combination of a confused deputy and a caller-sensitive method to achieve step (a) or (b). Consequently, modifying or replacing caller-sensitive methods like `Class.forName(String)`, `Class.getDeclaredMethods()`, and `Class.getDeclaredFields()` would render 22 out of 61 exploits infeasible.

In principle, acquiring instances of restricted classes does not require immediate use of a class loading vulnerability, or a confused deputy. An alternative way to get access to a restricted class is retrieving it from a trusted class. This is possible either because the trusted class leaks an instance through a public interface, or because it holds an instance in a private or protected

field, which can be accessed from untrusted code when using a vulnerability to break information hiding.

An example for such an attack is the exploit that involves CVE-2012-1726. It first uses a vulnerability to break information hiding, which provides access to private methods. Then, it uses a private method, `Thread.start0()`, to start a custom thread in such a way that this thread can retrieve an instance of `SunToolkit` using deserialization. `SunToolkit` is then used to access the contents of a private field in a system class, `AtomicBoolean.unsafe`, which holds an instance of `sun.misc.Unsafe`. Finally, `Unsafe` is used to define a class in a privileged context. As ca be seen, this vector does not require a class loading vulnerability or a confused deputy. However, this is a rather rare case, as only four exploits that use restricted classes implement a vector that does not depend on class loading vulnerabilities or confused deputies.

Multi-step attacks

The third category, multi-step attacks, is the smallest category and comprises the remaining two vectors, each implemented by a single exploit. They have in common, that they both abuse multiple vulnerabilities, but in contrast to the previous category of attacks, they do not immediately use restricted classes to achieve their final goal. Another common characteristic of the two exploits is that they both break information hiding.

One of the two exploits combines two different vulnerabilities to achieve this. The first one is used to get access to declared fields of a class, and the second one to set them accessible, so that field values can be read and written by the exploits. It uses this capability to set the private field `System.security` to `null`, thus disabling all security checks.

The second exploits combines a vulnerability that allows for loading arbitrary classes with another vulnerability that allows for calling arbitrary public methods. To implement a full attack, it first creates an instance of `java.beans.Statement`, which represents the method call `System.setSecurityManager(null)`. This alone does not violate the security model and cannot be used by untrusted code without causing an exception, because `Statement` will use the exploit's `AccessControlContext` to perform the call, which does not have permission to disable the security manager. To make use of this statement, the exploit first uses the class loading vulnerability to load `SunToolkit`. It then uses the second vulnerability to call the public method `SunToolkit.getField()`, in order to get access to the private field `Statement.acc` and set it accessible. This private field holds the instance of `AccessControlContext` that `Statement` uses to perform the method call that it represents. The exploit changes this field's value such that it holds a reference to another instance of `AccessControlContext` that has the permission to disable the security manager. After this modification, the `Statement` object can be successfully used to disable the security manager.

3.5 Discussion

The systematic in-depth study of the exploits in the sample set presented in this chapter revealed various different weak spots in the Java security architecture. One key finding is that the root cause of a large number of the different vulnerabilities that exploits abuse to implement their

attacks are design issues. Specifically, we can see that two major design flaws considerably impair the security of the Java runtime in practice.

The first design flaw we want to highlight is improper access control. Although Java's general approach of stack-based permission checks is not infallible, it is in principle an appropriate solution to implement access control. Remarkably, the issues exploited by attackers are not defects in the permission checks themselves, but primarily cases in which this mechanism is not properly used by system classes in the JCL. On the one hand, this concerns caller-sensitive methods in system classes that skip permission checks if the immediate caller is a trusted class by which they avoid proper stack inspection. This allows for confused-deputy attacks that can ultimately lead to a full bypass of all security mechanisms. On the other hand, this concerns the large number of restricted classes that entirely avoid the use of proper permission checks, while at the same time they implement security-sensitive functionality that yet again allows attackers to disable all security checks.

The second major design flaw besides improper access control is weak information hiding. It is particularly noticeable that the security of the entire Java runtime rests on the confidentiality and integrity of individual private fields and methods in system classes, while at the same time the runtime's mechanisms that shall enforce information hiding can be breached by individual implementation defects. Due to the lack of multiple safeguard mechanisms that backup each other, such vulnerabilities can all too easily be exploited by attackers.

The reflection API and method handles are features of the runtime that enable attackers to profit from both weak information hiding and improper access control. Although providing great flexibility and power to application developers, these features also present a major risk to the overall security of the runtime.

If mitigation strategies shall be developed to address the manifold issues we found in this study, one should consider that the design flaws discussed above interdepend. For example, confused deputies in combination with caller-sensitive methods that skip permission checks can be used to break information hiding, thus providing access to sensitive fields in system classes. In turn, a breach of information hiding can provide attackers access to methods in restricted classes that are not properly guarded by permission checks, which allows for, e.g., defining high-privileged classes. Both scenarios were implemented by exploits in the sample set and represent valid ways to illegitimately disable the Java runtime's security checks. We hence assume that systematic solutions to address both design flaws are required to improve the Java runtime's resistance against attacks.

Moreover, the large number of single-step attacks clearly indicates that defense in depth is needed to secure the platform, as individual implementation defects cannot be entirely ruled out when maintaining a complex software like the JRE.

Finally, we want to highlight that the weaknesses presented in this chapter may also be relevant to other platforms than Java. Specifically, certain weaknesses may also be relevant to Android, since most application code and system service code is written in Java. For instance, Peles et al. [87] showed that serialization vulnerabilities allow an attacker to execute arbitrary code in many Android applications and services which could result in privilege escalation.

3.6 Related work

To the best of our knowledge, this is the first study on a large set of Java exploits in which abstractions of exploits are compared to extract common patterns and identify weak spots in Java's security architecture. Moreover, we are not aware of similar studies performed for other platforms than Java.

There are publications describing how Java exploits work at a very low and technical level. For instance, FireEye [32] describes four Java vulnerabilities and Oh [78] studied a specific Java vulnerability that has been widely used for drive-by-download exploits. Kaspersky Lab [56] provide statistics on the attacks performed on Java regarding, e.g., the number of attacks over time, and the distribution of attacks in terms of geography. However, none of these publications provide a comparative systematic analysis of exploit behavior similar to the one presented in this chapter.

Schlumberger et al. [97] designed a tool to automatically detect malicious Java applets using a classifier produced by supervised machine learning. Their evaluation shows that this technique has the potential to detect even previously unknown malicious code. Approaches like these require a thorough selection of features, for which our extensive study of exploit behavior may serve as an input in future research.

In our study, we found that `Unsafe` is a restricted class that is frequently abused by exploits in our sample set. Mastrangelo et al. [63] empirically studied the use of `sun.misc.Unsafe` in benign real-world applications. They found that a large portion of the applications they analyzed use functionality in `Unsafe`, and they identified 14 different usage patterns. These patterns include performance optimizations, but also functionality not supported otherwise, such as updating the values of final fields.

Several improvements have been proposed to overcome limitations of the classical approach to stack-based access control (SBAC). Abadi and Fournet [5] proposed history-based access control (HBAC) which extends SBAC by not only considering the methods that are currently on the call stack to perform access control decisions, but also all methods that completed execution before the permission check has been triggered. Pistoia et al. [89] proposed information-based access control (IBAC) to improve over HBAC by properly selecting the methods that are actually responsible for a certain security-sensitive operation, thus making permission checks more restrictive and precise.

3.7 Conclusion

In this chapter, we presented a systematic and comprehensive study of a large body of Java exploit samples. As a first step, we harvested several online resources, such as exploit databases and a penetration-testing framework, which resulted in 87 findings. We reduced these original exploits to the minimal code needed to actually execute an attack and integrated them into our own custom exploit-testing framework. Then, we removed all exploits that were not reproducible from the sample set and merged multiple instances of the same exploit into one representation.

This resulted in a final set of 61 unique and reproducible minimal exploits. To the best of our knowledge, this is the largest sample set of Java exploits to date.

We developed a new meta model specifically for purposes of analyzing the behavior of a large body of exploits and used it to document the 61 minimal exploits. Based on this extensive documentation, we derived a set of nine weaknesses which comprise commonly used vulnerabilities and features of the Java runtime, e.g., unauthorized use of restricted classes, arbitrary class loading, caller sensitivity, or method handles. We explained in detail how attackers benefit from these weaknesses and how they can be combined to full attack vectors. Specifically, we found that there are three different categories of attacks: (1) single-step attacks (46% of all exploits), which exploit just a single vulnerability to achieve their final goal, (2) restricted class attacks (51%), which make use of a restricted class and combine multiple primitives to achieve their goal, and (3) multi-step attacks (3%), which use a combination of multiple vulnerabilities to disable all security checks.

Finally, we proposed ideas to improve the security architecture and discussed in more detail the underlying design flaws that enabled many of the attacks. This includes improper access control on the one hand, and weak information hiding on the other hand. Our study suggests that systematic solutions to address both these issues have the potential to substantially improve the security of the Java runtime in practice.

4

Hardening Java's information hiding[1]

In the previous chapter, we have investigated the various different attack vectors that attackers use to break the security guarantees of the Java platform. We identified that weak information hiding is one major design flaw in Java's security architecture. This chapter builds on this knowledge to develop a solution that systematically strengthens the Java platform against information-hiding attacks. As we will show, this hardening significantly increases the JRE's attack resistance by addressing that many exploits use security vulnerabilities to access private members of system classes in order to implement their attacks.

4.1 Motivation and contributions

Chapter 1 explains that the Java platform is widely deployed on desktop workstations and servers worldwide, and also highlights Java's popularity in the development community. However, we also explained that the JRE was subject to large-scale attacks which revealed many security vulnerabilities in the runtime. Between the years 2010 and 2018, the Oracle JRE has accumulated almost 600 CVE entries,[2] many of which represent high-critical vulnerabilities.

This high number of security defects is conspicuous, considering that Java's security architecture comprises various defense mechanisms, as we explain in Chapter 2. One might believe that Java's implementation of these mechanisms ensures that each mechanism is backed up by at least one other mechanism, however, the extensive study of Java exploits presented in Chapter 3 already revealed that severe design flaws weaken Java's security architecture in practice, thus allowing that individual security defects break the entire security architecture.

One of the flaws we specifically highlighted was weak information hiding. This fundamental language concept plays a major role in Java's policy enforcement, as the security status of the entire platform is primarily determined by the contents of private variables in system classes. One prominent example of such a security-critical field is `System.security`. It holds the instance of the security manager that is currently in charge of performing permission checks. Setting this field to `null` will disable policy enforcement, thus enabling arbitrary code execution. Another example of a security-critical field can be found in `java.beans.Statement`. Objects of this class represent arbitrary method calls. They can be instantiated at runtime and be

[1] Parts of this chapter were taken directly or with modifications from [50].

[2] `https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93`, accessed 05-Feb-2019

executed by calling `Statement.execute()`. To prevent untrusted code from gaining privileges by invoking a caller-sensitive method through `Statement` as opposed to calling it directly, `Statement` captures the current security context in a private field `Statement.acc` and ensures that `Statement.execute()` will call the target method only within this context. Using a breach of information hiding to replace the security context stored in `acc` by another context with all privileges allows attackers to elevate their privileges, as demonstrated by, e.g., the exploit for CVE-2013-2465.

Moreover, system classes store references to restricted classes in private fields. As explained in Chapter 2, many restricted classes implement functionality that can be used by attackers to disable policy enforcement. Since restricted classes often refrain from using stack-based access control to guard their methods, they are of high value to attackers. The application class loader prevents non-system classes from loading restricted classes, however, a breach of information hiding would allow an attacker to retrieve an instance of a restricted class from a private variable of a non-restricted system class.

As the above shows, information hiding is crucial to the security of the platform, and a failure to protect private members in system classes is a major security issue. At the same time, Java's approach to enforce information hiding is extremely brittle. In this chapter, we will show that individual, hard-to-prevent security defects are sufficient to break information hiding. For this reason, a systematic hardening of the Java runtime is required to strengthen the security architecture such that common attacks on information hiding will be blocked.

In the following we give detailed account of *two* systematic solution strategies to strengthen the platform. The first, the *heavyweight* mitigation strategy is clean and simple to explain, yet requires deep and extensive modifications of the JVM and its inner workings. Such modifications would require access to the JVM's source code, and significant engineering efforts to integrate our solution into Java runtimes of different vendors, on different platforms, in different versions, to support large-scale testing. This heavyweight solution is very useful for future platform implementation, but unfortunately precludes us from implementing and assessing this solution for the closed-source Java runtimes provided by Oracle and IBM.

Hence, to study the extent to which such attacks can be mitigated by a systematic solution that does not require modifications of the JVM, we also present a *lightweight* proof-of-concept approach that strengthens Java's information hiding by moving sensitive fields into a securely-encapsulated storage area, the "blackbox". This automatically renders impossible all accesses to those fields through previously exploited channels, for example the reflection API. The hardening further comprises a dedicated protection of private methods, classpool layout randomization, and integrity checks. Our implementation automatically integrates the proposed changes into past and current versions of the Java platform, without requiring access to its source code.

Our evaluation shows that 37 of the 61 exploits in the exploit sample set we presented in Chapter 3 break information hiding to implement their attacks, which highlights the importance of this subject. We further show that our proof-of-concept solution is highly effective—it blocks 31 of these exploits, as well as another exploit we added to the sample set specifically for purposes of evaluation. At the same time, the approach is backward compatible and retains high performance—all real-world applications we tested run unaffectedly, with an average performance overhead below 2%. As we also explain, if implemented, the heavyweight mitigation

strategy has the potential to outperform the lightweight proof-of-concept approach in terms of both speed and robustness.

This chapter presents the following core contributions:

- A systematic analysis of Java's information hiding, which highlights systematic flaws in Java's security model.
- A systematic hardening of Java's information hiding as a proof-of-concept implementation.
- An analysis showing that the proof of concept blocks 84% of past information hiding exploits, while retaining backward compatibility and high performance.
- An elaborate solution approach to strengthen information hiding for productive use, which improves over the proof of concept.

4.2 Threat model

We outlined the fundamentals of the Java security architecture in Chapter 2 and explained that multiple security mechanisms are combined to achieve the security guarantees provided by the Java platform, such as type safety. Interestingly, however, these mechanisms do not implement the concept of defense in depth, but rather complement each other, which results in the fact that each mechanism resembles a single point of failure—all mechanisms *interdepend*, but they do not *backup* each other. Hence, a single security defect in any of these mechanisms, such as bytecode verification, or automatic memory management, can be sufficient to break all security guarantees of the entire Java platform. Information hiding is especially brittle, as it requires multiple security mechanisms to properly work together. This includes bytecode verification to prevent illegal static references, permission checks in introspection APIs to prevent dynamic access to sensitive variables, automatic memory management to prevent manipulation of field values stored on the heap, and numerous other components.

One must acknowledge that a bug-free implementation of all these complex mechanisms is impossible with reasonable effort and state-of-the-art technology. But for this reason, it was argued by Pompon [91] and others earlier that in a secure software design one must "assume breach". The different security mechanisms that constitute a security architecture should hence be designed to provide defense in depth, which assures that a defect in one of those mechanisms can only have a restricted impact on the overall security of the system.

4.2.1 Attacker capabilities

The attacker's primary goal in our threat model is to escalate privileges of attacker-controlled code executed by a Java runtime running on the target machine with security manager enabled. For this, the attacker breaks information hiding by exploiting security vulnerabilities in the Java runtime to gain access to sensitive private variables and/or methods in system classes, which in turn are used to achieve privilege escalation.

For the defenses we present in this chapter, we assume that the attacker has the following capabilities. The presented defenses are designed to withstand attackers with those capabilities.

- Execute Java code with a restricted set of privileges, or even no privileges, on the target machine's Java runtime.

- Exploit vulnerabilities in the Java runtime that allow for illegitimate access to private sensitive variables and methods in system classes. Section 4.2.2 presents the various attack vectors that can be used for exploitation. We showed in Chapter 3 that Java exploits make use of exactly such vulnerabilities.
- Statically and dynamically analyze the installation files of the Java runtime as they are deployed by the vendor, e.g., in the form of a setup binary offered to download on a website, or an installation package offered in a package repository.

We further assume that the attacker has the following restrictions.

- The attacker cannot intercept method calls between system classes. This includes that call arguments provided by system classes and return values received are confidential and cannot be modified by attackers. Also, the attacker is incapable of preventing method calls between system classes. This is a realistic assumption unless the system has been fully compromised already.
- The attacker cannot modify the Java runtime on the target machine in order to prepare the attack. The binary representations of all system classes and native components of the Java platform on the target machine are out of reach for the attacker. This is also a realistic assumption unless the system has already been compromised otherwise.

The original security architecture of the Java platform already depends on the above restrictions.

4.2.2 Attack vectors to break information hiding

We reconsidered the 61 minimal exploits presented in Chapter 3 and reviewed them with a specific focus on breaches of information hiding. We found that 37 of the exploits in this sample set directly break Java's information hiding, for which they are using one of the following three ways. Interestingly, each of these attack strategies can be implemented by exploiting just a single security vulnerability, which supports our claim that the Java security architecture systematically lacks defense in depth.

Attack vector 1: Insecure use of introspection
The reflection API and method handles allow for class introspection at runtime. Any code can use public interfaces of these APIs to access fields, or to invoke methods. Due to the sensitive nature of such functionality, introspection APIs implement dynamic access-control checks to prevent illegal access to sensitive members, for example, if an application class attempts to access a private member of a system class. Generally, such access-control checks are implemented through stack-based access control—relevant methods in, e.g., the reflection API are guarded by a call to the security manager, which ensures that all code involved in the action has been assigned appropriate privileges. For example, the `java.lang.reflect.ReflectPermission` with argument "suppressAccessChecks" must be granted to classes that attempt to reflectively access non-public class members of other classes.

There are various kinds of vulnerabilities that allow attackers to misuse introspection APIs in order to bypass Java's rules for information hiding.

One problem is that Java's implementation of introspection APIs is complex, and also scattered across multiple parts of the JCL. It is thus hard to ensure that all execution paths through these APIs are guarded by appropriate permission checks that prevent illegitimate access from one class to private members of another class. A study by Srivastava et al. [102] previously revealed several instances of incorrect or missing checks in different implementations of the JCL, which shows that implementing policy enforcement is an error-prone task. If attackers detect one path in an introspection API that provides unconstrained reflective access to private class members, information hiding can be bypassed.

Another common problem is that trusted system classes in the JCL can use introspection APIs in an insecure manner, which turns them into confused deputies. This is always the case if a system class allows untrusted code to determine the target of reflective access, e.g., a system class that calls arbitrary method handles originally provided by untrusted application code, or field accesses to fields whose names are provided by untrusted code. In these scenarios, attackers can profit from confused-deputies that incautiously implement reflective access in privileged blocks, which allows them to bypass permission checks in the reflection API, or they simply profit from caller-sensitive target methods that behave differently because their immediate caller is no longer untrusted application code, but instead the trusted confused-deputy.

Attack vector 2: Type confusion

Java was designed to provide strong type-safety guarantees, which restricts the way that code can interact with objects and memory, thus contributing to strong information hiding. The platform combines static and dynamic checks to prevent that an object of one type is illegally cast to an incompatible type. Bytecode verification, for example, statically checks assignment compatibility in method bodies during class loading to a certain extent, and ensures that `checkcast` instructions are present in places where static checks cannot be performed, so that type safety is dynamically enforced by the JVM in these cases [62, Sec. 4.10.1]. However, enforcing type safety is complex, and in the past, several defects made the platform susceptible to so-called type-confusion attacks.

In a type-confusion attack, the exploit causes the JVM to incorrectly treat an object of a certain type `A` as an object of another type `B`. Consequently, the attacker can perform actions on this object that are allowed for type `B`, but not for type `A`. If, for example, type `A` declares a private field at a certain offset, and type `B` declares a public field at the same position, type confusion can be used to illegally access the private field—convenient, for instance, inf one wishes to null out the field `System.security`.

Attack vector 3: Buffer overflows

The Java platform provides a level of abstraction between the hardware and application layer. Java applications do not have to deal with clean memory allocation and deallocation, or a thorough distinction of code and data. However, the platform itself is partially implemented in native code and hence potentially susceptible to buffer overflow vulnerabilities.

A buffer overflow vulnerability is a software defect that occurs when an application writes more data into a target buffer than it was originally supposed to handle. In consequence, the write operation exceeds (or *overflows*, figuratively) the target buffer's boundaries and adjacent memory locations will be overwritten in the action. In and by itself, the buffer overflow is a local

breach of integrity, not confidentiality, as attackers can use the buffer overflow to manipulate certain parts of memory, but not to read memory.

The consequence of overwriting memory locations this way is different from case to case, and strongly depends on the type of memory locations that are affected by the vulnerability. If the target buffer that is affected by the vulnerability resides on the stack,[3] attackers will typically attempt to overwrite return addresses, which would allow them to take over the control flow of the application. If instead the vulnerable buffer is allocated on the heap,[4] attackers might attempt to manipulate application-specific data structures, as the stack is out of reach. In the context of Java security, this might be the contents of sensitive variables of system classes which are stored on the heap.

Due to the different ways in which different buffer overflow vulnerabilities can impact the security of the Java platform, we differentiate two different cases. *Information-hiding attacks* use buffer overflows that affect buffers that are near the locations of sensitive variables of system classes, which the JVM stores on the heap. Such attacks can be used to overwrite private variables and hence represent a breach of information hiding. One example of such an attack on the Java platform is the exploit for CVE-2013-2465. These attacks are within our scope and the countermeasures presented in this chapter are designed to mitigate them.

Control-flow hijacking attacks use buffer overflows that affect memory locations whose integrity is important to the control flow of the application, such as return addresses on the call stack of the JVM. These attacks can take over the application's control flow and hence achieve arbitrary code execution without corrupting information hiding. Control-flow hijacking attacks are not specific to Java, but instead affect a broad range of native applications. There are a variety of countermeasures that have been implemented in the past, such as data execution prevention [103], or address space layout randomization, which mitigate, or at least complicate control-flow hijacking attacks. Moreover, there is an active research community that develops and improves novel techniques to mitigate such attacks, e.g., by enforcing the control-flow integrity of native applications [3]. Note that the JVM is equally susceptible to control-flow hijacking attacks as, e.g., PDF viewers, browsers, or other native applications. This means that a solution that mitigates such attacks for Java is likely a general solution to the problem. Special attention is required because the JVM implements a just-in-time compiler that generates new code during program execution. This is challenging because the runtime deliberately turns data into executable code, which can even happen multiple times for the same code during a single application run, e.g., to apply different levels of optimization. However, just-in-time compilation is commonly implemented in many applications that process user-provided scripts, such as web browsers, which is why the research community has already proposed dedicated solutions strategies to prevent attacks that involve just-in-time compiled code [15].

Since the focus of this chapter is on strengthening information hiding in Java, we consider control-flow hijacking attacks as out of scope. However, the countermeasures that were proposed in academia to address the problem of control-flow hijacking can also be implemented for the JVM, even complementary to the countermeasures we propose in this chapter to strengthen information hiding.

[3] see "CWE-121: Stack-based Buffer Overflow"
[4] see "CVE-122: Heap-based Buffer Overflow"

4.3 Proof-of-concept solution

As explained in Section 4.1, this chapter gives detailed account of *two* possibilities to strengthen information hiding such as to avert common attacks on information hiding. The first, the *heavyweight* mitigation strategy is clean and simple to explain, and is what we would suggest for productive use. Yet, implementing this solution requires access to the JVM's source code, and significant engineering effort. We will explain the design of this strategy in Section 4.5.

In this section, we instead present a backward compatible prototype solution that goes without modifications of the JVM. It is composed of a systematic orchestration of several countermeasures specifically designed to strengthen information hiding in Java. These countermeasures effectively block a broad range of attacks on information hiding, while retaining backward compatibility and high performance.

We specifically designed this prototype such that it can be implemented at a cost that does not have to be borne by a large development team, and such that it can be integrated into even closed-source JREs. Conducting experiments with real-world applications and exploits allows us to study in-depth how information hiding contributes to Java platform security, and to what extent—and at what cost—information-hiding attacks can be blocked. The design we chose allows us to perform experiments on various platforms for which we have no access to the source code. Using multiple platforms for testing is important, as there are significant differences between the implementations of the different runtimes. For example, from testing we know that there are applications that run on the Oracle JDK, but cause errors on the OpenJDK. Also, the sample set of exploits presented in Chapter 3 that we also use in this chapter for information gathering and evaluation contains attacks that affect only specific JDKs.

In summary, the key strengths of our prototype implementation are:

- To apply the countermeasures, no access to the Java platform's source code is required. All modifications are automatically integrated into the target platform's JCL via instrumentation, no modifications to native components of the JRE are required.
- The prototype is generally platform-independent. It can be applied to various different versions of Java, even by different vendors, running on different operating systems.
- In comparison to the heavyweight solution for productive use, it requires lower implementation effort.

However, as we explain in detail below, these advantages come at the cost of using secret tokens that our proof of concept hardwires into the method bodies of system classes. As we explain, these tokens are randomly generated and individual for every host system. It would be possible to generate and update the secret tokens during the installation routine of the JRE, or even before every single application run. Alternatively, it would also be possible to modify the Java runtime's class loading mechanism in such a way that it automatically generates and integrates new secret tokens into system classes while they are loaded by the JVM. This way, secret tokens would never be persisted on disk, which would further complicate attacks.

For our proof-of-concept solution, we thus have to extend the threat model presented in Section 4.2 by the following restriction: *Attackers cannot inspect or analyze the behavior of Java system classes on the target machine.* This restriction forbids that attackers learn secret tokens by analyzing method bodies of system classes on the target machine. Note that analyzing

Table 4.1: Relationship between countermeasures and attack vectors

	Attack vectors		
Countermeasure	**Introspection**	**Type confusion**	**Buffer overflow**
Field blackbox	✓	✓	✓
Integrity checker	(✓)*	(✓)*	(✓)*
Method blackbox	✓	✓	
Classp. layout random.		✓	✓

*partially: ensures field values' integrity, but not confidentiality

the installation files of the the JRE deployed by the vendor is not an issue, as secret tokens are randomly generated on the target machine. The additional restriction is certainly realistic for systems that have not already been fully compromised—the JRE systematically prevents code from inspecting method bodies of system classes. The JCL provides no dedicated functionality to inspect method bodies, and even exceptions or uncontrolled crashes reveal no details about the contents of methods to potential attackers.

The solution we propose for productive use in Section 4.5 does not depend on any secret tokens, and hence requires no extensions to the threat model presented in Section 4.2.

4.3.1 Conceptual overview

Section 4.2.2 presents the various attack vectors that attackers can use to break information hiding: insecure use of introspection, type confusion, and buffer overflows. To address these attacks, we propose the "blackbox": a combination of four countermeasures that can be integrated into the JCL to prevent illegal member access. The following details these countermeasures and their individual purposes in defending information-hiding attacks. Table 4.1 provides an overview of how the countermeasures relate to the different attack vectors.

1. *The **field blackbox** restricts sensitive fields to be accessed only from known, trusted code*—Its purpose is to ensure the confidentiality and integrity of field values. Java runtimes commonly store field values along with their parent object's representation in memory, thus enabling low-level attacks by means of type confusion, or buffer overflows, by which attackers gain access to these sensitive variables from exploit code. Further, attackers can use introspection APIs to dynamically access sensitive fields in system classes at runtime if not protected properly. The blackbox prevents illegitimate access to these fields by storing their values in an isolated storage area to which introspection APIs have only limited access. As we explain, this countermeasure ensures that sensitive fields in system classes can only be accessed by system classes, which can be trusted as they were provided by the vendor.
2. *The **method blackbox** restricts sensitive methods to be accessed only from known, trusted code*—Its purpose is to prevent exploits from calling private methods of system classes using introspection APIs or type confusion. Specifically, this countermeasure ensures that

sensitive private methods in system classes can only be called by their respective declaring class, which is trusted as it was provided by the vendor.

3. *Some sensitive fields are read (never written!) directly by the JVM. The **integrity checker** assures their integrity, without requiring JVM modifications*—To this end, it adds additional integrity checks to the JCL that serve as security gates, illegal field modification is thus detected before it impacts the security of the platform. In contrast to the field blackbox, the integrity checker's purpose is to merely ensure the integrity of sensitive fields, not their confidentiality. Consequently, it is applied only to fields whose confidentiality is not important for the security of the platform. This countermeasure is entirely obsolete in scenarios in which it is acceptable to modify the JVM.
4. ***Classpool layout randomization** is an additional defense layer against type confusion and buffer overflows*—It randomizes the location of field values in memory by modifying the classpool in class files of system classes. Past exploits using type confusion or buffer overflows to access fields commonly expect target field values to be stored at the same locations. Classpool randomization breaks these assumptions and thus blocks basic attacks. It may be bypassed by search routines that locate field values in memory, so it alone is not sufficient as a countermeasure. However, it complements the field blackbox and integrity checker by adding an additional layer of security, thus raising the bar for future attacks, and contributing to defense in depth.

4.3.2 Design

This section discusses relevant design considerations of our proof-of-concept implementation.

Field blackbox

As many different attack vectors rely on illegal access to private fields in system classes, protecting sensitive fields is a key feature of the blackbox. We consider all private fields in system classes of the following types as sensitive: `Unsafe`, `SecurityManager`, `AccessControlContext`, `ProtectionDomain`, and `PermissionCollection`, including their array representations. We derived this list of relevant types from a thorough review of the Java security architecture [39], as well as from reviews of exploits in our sample set, see Chapter 3.

To protect sensitive field values, we replace all regular accesses to them by calls to the blackbox, thus ensuring that they will never be read or written from their original locations. This protects against type confusion and buffer overflows, because sensitive values are no longer stored along with the parent object's representation in memory, but rather in a hard to access location in the blackbox. Conceptually, the blackbox can use various measures to protect its contents, including, but not limited to, memory randomization, on-the-fly encryption, process separation, and even hardware separation. The CHERI ISA [109], for example, is an instruction set that provides dedicated support for hardware-based, fine-grained memory protection. Using the CHERI ISA allows applications to securely restrict how code can access certain memory locations, which could be used by a blackbox implementation to prevent illegitimate access to the values it contains. As we explain in Section 4.6, Chisnall et al. [17] applied the CHERI ISA

Listing 4.1: Effects of integrating the field blackbox into the JCL

(a) Field access in a system class without modifications

```
1  public SystemClass {
2    // sensitive field
3    private static AccessControlContext acc;
4
5    // original code
6    public void sysMethod(AccessControlContext newAcc) {
7      acc = new Acc;
8      System.out.println(acc);
9    }
10 }
```

(b) Access to the field blackbox in a modified system class

```
1  public SystemClass {
2    // sensitive field - now just an empty placeholder
3    private static AccessControlContext acc;
4
5    // modified code
6    public void sysMethod(AccessControlContext newAcc) {
7      Blackbox.set_static_ref(3384, 78832498L, newAcc);
8      System.out.println(Blackbox.get_static_ref(3384, 78832498L));
9    }
10 }
```

to implement a sandboxing mechanism for native code in Java, which highlights the relevance of this instruction set for Java platform security.

Listing 4.1 shows the effects of integrating the field blackbox into the JCL. Listing 4.1a shows how unmodified system classes access sensitive field values. In this example, `SystemClass` declares a private field `acc` of type `AccessControlContext`. The method `sysMethod()` reads and writes directly the static field in its parent class. After integrating the field blackbox into the JCL, rather than loading and storing values in `acc`, `sysMethod()` uses the blackbox's public interface to read and write sensitive values, as can be seen in Listing 4.1b. To access field values in the blackbox, `sysMethod()` passes at least two arguments. The first argument, here 3384, is a secret token required to access *any* security-relevant functionality in the blackbox. Importantly, the runtime automatically protects the secret token much better than the original sensitive field, as it is hardcoded into the method body of the system class, which cannot be inspected or analyzed by untrusted code. The secret tokens cannot be guessed: Any attempt to access a blackbox method without the correct token is considered an attack, and will result in a call to `Blackbox.panic()`. This method is an emergency method that adequately responds to an attack, e.g., by generating logs and reports for manual interventions, or even by terminating the execution of a violating application. Note that the solution we propose in Section 4.5 to strengthen information hiding for productive use does not depend on such secret tokens. Instead,

it extends the JVM instruction set by instructions that can only be used by system classes, which, however, requires modifications to the JVM. The second argument, here 78832498L, is a unique identifier associated with the field that is accessed in the original code. The blackbox provides a small set of public interface methods to handle large numbers of different sensitive values, unique identifiers are thus required to map field accesses to different fields in the original code to specific storage locations in the blackbox in a semantically-equivalent manner.

Replacing sensitive field access in system classes as illustrated in Listing 4.1b effectively causes sensitive fields such as `acc` to become "dead" as they will not any more be read or written by their parent class. Without any further modifications, this would cause legacy reflective accesses to such fields to misbehave, because the reflection API would access the original, now "dead" field itself, while the variable's parent class would use the field blackbox instead.

Hence one should seek to support reflective accesses to the blackbox. Yet, while doing so, one must be mindful of the problem of reflection-based confused-deputy attacks. Confused-deputy vulnerabilities are caused by system classes that make insecure use of reflection—they call methods of the reflection API in a way that allows untrusted code to arbitrarily specify the targets of reflective access, thus providing access to private variables of system classes. `SunToolkit` is an example of such a confused deputy. In past versions of Java, it contained a method `getField(targetClass, targetFieldName)` that explicitly elevated privileges through a call to `doPrivileged()` to be able to access and return any target field specified by the caller. A secure use of reflection, in contrast, is performed using constant arguments, which cannot be modified by untrusted code. In the JCL, we found actual examples of system classes that make secure use of reflection to access sensitive fields of other system classes. Preserving this legitimate access is thus required to retain backward compatibility. We did not find any examples of classes in the JCL which make insecure use of reflection to access sensitive fields of other system classes.

We thus propose to extend the reflection API and modify call sites accordingly such that secure, i.e., restricted, uses of reflection will be automatically rerouted to the field blackbox, while insecure, i.e., unrestricted, uses of reflection remain unchanged. To detect relevant call sites, we statically analyze all system classes in the JCL to find calls to `Class.getDeclaredField(fieldName)`. Since variables whose values are stored in the blackbox remain in their parent classes as structural placeholders, `getDeclaredField()` continues to function without any modifications. For every such finding, the analysis determines if the call is performed on constant values for `Class` and `fieldName`. As the attacker cannot modify such calls, they imply a restricted, secure use of reflection. Actions performed on an instance of `Field` obtained in this secure manner can be safely rerouted to the field blackbox. To achieve this, the analysis instruments any secure call site for `getDeclaredField()` such that the returned instance of `Field` will be permanently associated with the unique identifier for the concerning field. This identifier is required to access the corresponding value in the field blackbox. Our modification of the reflection API causes methods such as `Field.get()` to check for such an association of a unique identifier. If such an associating exists, rather than attempting to read the respective value from the original variable, the identifier is used to obtain it from the field blackbox. If the static analysis finds a call site that it does not consider to be secure, or it detects a secure call site that targets a non-sensitive field, the original code will not be modified and the method call

Listing 4.2: Secure reflective access to sensitive fields will be rerouted to the blackbox

```
1  public Unsafe secReflect() {
2    Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
3    unsafe.enableBlackbox(3384, 78932788L);
4    unsafe.setAccessible(true);
5    return (Unsafe)unsafe.get(null); // succeeds
6  }
7
8  public Unsafe insecReflect(String fieldName) {
9    Field unsafe = Unsafe.class.getDeclaredField(fieldName);
10   unsafe.setAccessible(true);
11   return (Unsafe)f.get(null); // fails
12 }
```

will not be rerouted to the blackbox. Attackers thus cannot access sensitive values stored in the blackbox by abusing such reflective calls. SunToolkit can no longer be used for attacks

Listing 4.2 illustrates how the analysis deals with different uses of reflection. In this example, `secReflect()` invokes `getDeclaredField()` in line 2 on constant values `Unsafe.class` and `"theUnsafe"`, which the analysis considers to be a secure use of reflection. Because of that, and the fact that `Unsafe.theUnsafe` is a sensitive variable, the analysis will add a call to `enableBlackbox(token, uid)` in line 3 to associated the returned instance of `Field` with the unique field identifier for `Unsafe.theUnsafe`, here 78932788L. The first argument, here 3384, is the secret token required to access functionality in the blackbox, which prevents illegal access. Due to our modification of the reflection API, `Field.get()` will check for such an association, and query the field blackbox to get the desired value, rather than reading it from the original variable. The call in line 5 will thus succeed and return the correct value. As explained before, the solution we propose for productive use in Section 4.5 does not use any secret tokens to prevent illegitimate calls, it rather extends the JVM instruction set by instructions that only system classes can use.

The second method in this example, `insecReflect()`, is treated differently. In contrast to `secReflect()`, it contains a call to `getDeclaredField()` in line 9 that passes a variable as field name, rather than a constant. The analysis considers this insecure and does not modify the call site to reroute `Field.get()` in line 11 to the blackbox. Consequently, an attempt to get a sensitive value this way will fail, because the reflection API will not query the field blackbox, but instead attempt to read the value from the original variable. This variable, however, contains `null`, because its parent class uses the field blackbox to store sensitive values.

The solution, as presented above, only reroutes restricted, secure uses of reflection to the blackbox. Any attempt to access a sensitive value using method handles, however, would fail, because we did not design an equivalent modification of the method handles API. The reason for this is that we are not aware of any system classes that would require such access. This is not at all surprising, as method handles are a comparably new feature, and significant portions of the

Listing 4.3: Integrating the method blackbox into system classes

(a) Private method call in a system class without modifications

```
1  public class SystemClass{
2    // security-sensitive private method
3    private Field[] getPrivateFields(Class c) {
4      return filterPrivate(c.getDeclaredFields());
5    }
6
7    // publicly accessible method
8    public boolean hasPrivateFields(Class c) {
9    if(getPrivateFields(c).length > 0)
10     return true;
11   else
12     return false;
13   }
14 }
```

(b) Private method call in a modified system class

```
1  public class SystemClass {
2    private Field[] getPrivateFields(int token, Class c) {
3      if(token != 4853233) {
4        Blackbox.panic();
5      }
6      // ... original code
7    }
8
9    public boolean hasPrivateFields(Class c) {
10     if(getPrivateFields(4853233), c).length > 0)
11     // ... original code
12   }
13 }
```

JCL are in fact older. If such support should be needed in the future, we expect that a similar approach like the one we presented for the reflection API can be implemented.

Method blackbox

Certain private methods in system classes can be used to disable security checks or compromise the platform through other means. Regardless of how exploits obtain reflective access to such sensitive methods, at some point they have to call them to actually make use of them. The purpose of the method blackbox is to prevent attackers from using type confusion or reflection to call these sensitive methods. As previously explained in Section 4.2.2, we consider out of scope such attacks that use buffer overflows to take over the control flow of the application.

As illustrated in Listing 4.3 the method blackbox is simple, and yet effective against common attacks. In this example, `getPrivateFields()` is a sensitive private method of `SystemClass`

that must be protected from illegal invocation. To achieve this, the method blackbox adds an additional argument `token` to `getPrivateFields()` that will be checked at the beginning of the method, see lines 3–5 in Listing 4.3b. The method will continue its normal operations only if `token` contains a specific value, here 4853233, otherwise, `Blackbox.panic()` will be called, which serves as an emergency method that adequately responds to attempted attacks. As can be seen in line 10 in Listing 4.3b, all callers in the parent class will be updated such that they pass the required secret token as an argument, backward compatibility is thus retained. Exploits that obtain access to methods protected this way, e.g., by using a confused deputy, cannot call them without the correct token. Importantly, the solution we propose for productive use in Section 4.5 does not use any secret tokens. Instead, it extends the JVM instruction set by dedicated instructions for sensitive method calls that only system classes can use.

We currently apply this countermeasure to all private methods in `java.lang.Class` with return types `Field`, `Method`, or array representations of these types, because we found indication that some of these methods are used by exploits to get illegal access to method or field handles. We did not encounter any reflective access to one of these methods in system classes, nor did we find any direct calls from the JVM. If reflective access to methods protected by the method blackbox ought to be retained, it is possible to apply the same approach that is used to deal with reflective access to fields protected by the field blackbox.

Integrity checker

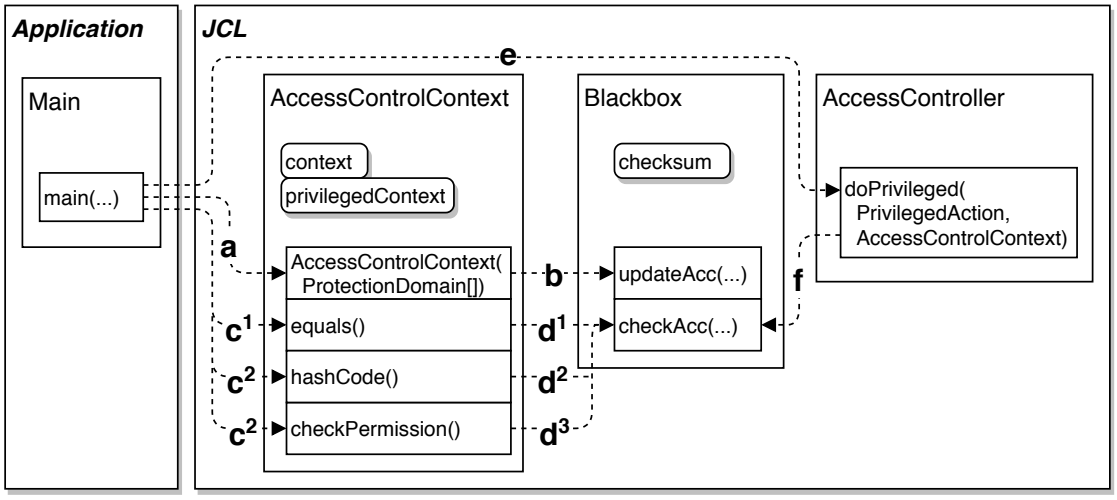
The field blackbox is the core feature of our proposed solution, but there are some sensitive fields in the JCL to which it cannot be applied. Those fields are accessed not only by their parent class, but also directly by native code in the JVM. Replacing field accesses only in their parent classes while leaving accesses in the JVM unmodified would result in inconsistent behavior. However, modifying the JVM accordingly would require access to its source code, thus violating the design goals of our proof of concept (see Section 4.3).

A thorough review of the relevant fields reveals that in all cases the confidentiality of their values is not an issue, but only their integrity is important to the security of the platform. The variable `AccessControlContext.context` of type `ProtectionDomain[]` is an example of such a field. It contains the set of protection domains that will be considered when a permission check is to be executed in the respective context. The protection domains are not secret and could possibly even be obtained through other means, so it is not required to protect the confidentiality of `context`. Its integrity, however, is of high importance, because illegal modification would compromise access control.

Further, we found that the JVM writes to these fields only during object initialization, if at all. In all other cases, the JVM only performs read accesses, e.g., reading `AccessControlContext.context` when a permission check is to be executed.

To protect also these special fields, we developed the *integrity checker*, an alternative countermeasure that is used to guard sensitive fields that cannot be moved to the blackbox. The basic concept of the integrity checker is to track throughout program execution all legal field modifications of protected fields such that illegal modifications can be detected before they impact the security of the platform. To achieve this, we keep a checksum in the field blackbox for every object that contains at least one field whose integrity is to be protected. Every legal field

Figure 4.1: Integrity checks in the JCL prevent illegal field modification



modification in a parent class will cause the corresponding checksum in the field blackbox to be updated. To prevent attackers from simply updating themselves the checksum after illegal field modification, updating this value requires a secret token known only to the parent class. Similarly to the field blackbox, this token is stored only within the class' code.

We added integrity checks to the JCL that serve as security gates—whenever they are reached during program execution, they will evaluate for relevant objects whether the current field values of protected fields still correspond to the checksum in the field blackbox. If this is not the case, at least one of the fields must have been modified illegally, and the blackbox will call its emergency method `Blackbox.panic()`, which adequately responds to attempted attacks. If the checksum matches, and no illegal modification has been detected, the check will return silently. On the one hand, such security gates are added to all public methods of classes that contain at least one field that is to be protected by the integrity checker. Before a public method reads one of the protected fields for the first time, an integrity check will be performed by comparing checksums. On the other hand, we add security gates to call sites that depend on the integrity of a specific object involved in the action, even though no method of the object is called at that point. For example, we add checks before all calls to `AccessController.doPrivileged(PrivilegedAction pa, AccessControlContext acc)`, because the integrity of `acc` is important in the context of privilege elevation. The call sequence initiated by `doPrivileged()` does not contain any calls to methods of `acc`, but it will cause the JVM to directly read the contents of `acc.context`. Illegal modifications of this value thus have to be detected before the operation is executed.

Figure 4.1 is a simplified illustration that shows how the blackbox is used to ensure the integrity of field values in `AccessControlContext`. This class contains two special fields whose integrity must be protected: `context` of type `ProtectionDomain[]` and `privileged-Context` of type `AccessControlContext`. We explained above that we consider fields of these types as sensitive variables. In the example illustrated in Figure 4.1, an application's `main()` method calls the constructor `AccessControlContext(ProtectionDomain[])` to in-

stantiate `AccessControlContext` (arrow "a"). This constructor writes to at least one of the two sensitive fields. Because of that, our prototype modified this constructor such that, after modifying these fields, it calls `Blackbox.updateAcc()` to update the checksum that is kept in the blackbox specifically for this instance of `AccessControlContext` (arrow "b"). Subsequently, the application calls a set of public methods in `AccessControlContext` which read the values of `context` or `privilegedContext` (arrows "c^1"–"c^3"). Our prototype added security gates also to these methods so that they call `Blackbox.checkAcc()` to ensure the integrity of the two sensitive fields before they perform their regular tasks (arrows "d^1"–"d^3"). Finally, the application calls `doPrivileged(PrivilegedAction, AccessControlContext)`, which, as we explained above, depends on the integrity of `AccessControlContext.context` (arrow "e"). Because of that, we also added a security gate at the beginning of this method that calls `Blackbox.checkAcc()` (arrow "f"). Taken together, the security gates added to all public methods of `AccessControlContext`, and additional relevant methods like `doPrivileged(PrivilegedAction, AccessControlContext)` ensure that illegal field modifications of `context` or `privilegedContext` are detected before they impact the security of the JRE.

The relevance of call sites like `doPrivileged()`, which depend on the integrity of fields that are declared in other classes and are also directly accessed by native code in the JVM could in theory be determined automatically, e.g., by statically analyzing the source code of the JCL and the native code of the JVM. However, implementing such an analysis would be challenging, as the analysis would have to be designed as a hybrid solution that can handle native code and Java code at the same time. The analysis would be even harder to implement for closed-source JREs, as analyzing binary code is a complex task in itself. Considering these major issues, we decided to collect relevant call sites manually at this stage of development. This is sensible, as the number of fields protected by the integrity checker is low.

Classpool layout randomization

Classpool layout randomization is a countermeasure specifically designed to break backward compatibility with all existing exploits that illegally access fields in system classes through type confusion, or other attacks that make assumptions about field locations in memory.

In a type confusion attack, the attacker holds a reference to an object in memory that is associated with different type information than the actual object in memory. Accessing the object using a spoofed reference can thus be used to bypass visibility restrictions imposed by the actual type of the target object in memory. If, for example, the target object defines a private field at a certain location, and the spoofed type defines a public field at the same location, the spoofed reference can be used to access the private field of the target object, because the Java runtime will check field visibility for the wrong type.

Every class file contains a section known as the *classpool*, which lists all class and instance fields of the type. The location of fields in memory is dependent on their position in the classpool. Past exploits expect certain sensitive fields to be at certain memory locations, because their location in the classpool is known.

To break assumptions of known exploits and increase the effort required to implement new low-level exploits, we apply classpool layout randomization. This countermeasure shuffles the

order of existing fields in the classpool and adds a random number of new dummy fields, which results in randomized locations of field values in memory. All attacks that aim for specific fields in memory will thus have to integrate a search routine to find the correct locations. We currently apply this countermeasure only to `AccessControlContext`, but it can be applied to other system classes as well.

4.3.3 Implementation

To be able to validate our prototype implementation of the proposed countermeasures on different JREs, we designed it as a transformation engine. It takes as input the JCL or a given JRE (e.g., the `rt.jar` file of the Oracle JRE), then modifies all required classes to integrate our proposed countermeasures, and finally outputs the set of files that have changed. Then any original, unmodified JVM can be instructed via command line arguments to use the modified system classes that were generated by the transformation engine.

Our implementation uses the Javassist[5] library to locate fields and methods, and to implement all modifications to class files that are necessary to integrate our proposed countermeasures into the JCL. A cryptographically strong random number generator provided by `java.security.SecureRandom` is used to generate all secret tokens. Further, we use the ASM[6] library to identify system classes that implement statically-confined reflective accesses to fields protected by the blackbox, so that we can redirect these calls accordingly, see Section 4.3.2. In contrast to the manipulation of native binaries, it is significantly less involved and error-prone to analyze and modify Java class files. The class file format is well defined and all class files comprehensively provide structural information about the Java classes they represent, including a list of declared fields, their types and symbolic names, as well as a list of methods, method names and signatures. We have not experienced any issues with locating class members or modifying method bodies.

The exact number of class members that must be protected by the blackbox is slightly different for each JRE. For the Oracle JRE 7, the engine transforms approx. 180 files to integrate the field blackbox, ten methods are protected by the method blackbox, three fields are protected by the integrity checker, and one class is protected by classpool layout randomization. The transformation engine is implemented such that it automatically searches for fields and methods that meet our specified criteria, which significantly reduces the effort required to adjust the engine to previously unseen JRE implementations. This high portability allowed us to apply it to Oracle Java 7 and 8 on Windows, as well as to IBM Java 7 on Linux.

The prototype features two modes of operation. In one of them, all sensitive values are stored in native code, all accesses have to be made through JNI. Since this is implemented in native code, the technical measures that could be used to protect the blackbox and its contents are unlimited. In the other mode, sensitive values are stored in a dedicated Java class, which is only used within methods of system classes that legally access the blackbox. This mode of operation is highly portable as it is implemented in Java, but its capabilities to use certain protection mechanisms are also restricted.

[5] `http://www.javassist.org`
[6] `https://asm.ow2.io`

Figure 4.2: JVM memory layout in modified JRE

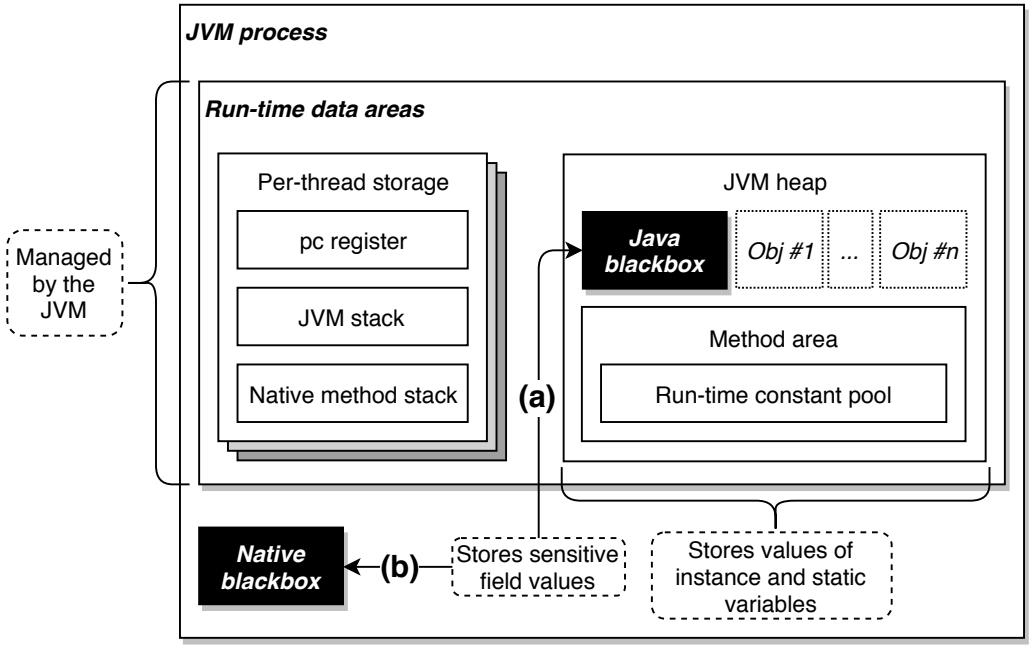


Figure 4.2 illustrates the JVM's memory layout in these two settings, assuming an implementation in accordance with the Java Virtual Machine Specification [62]. If the Java blackbox mode of operation is used (see (a)), all sensitive field values are still stored on the JVM's heap. However, the difference to original JVM implementations is that now sensitive values are no longer stored along with the parent object's representation in memory, but rather in the object representation of the blackbox class. This prevents that vulnerabilities that result in unsafe memory operations on specific Java system classes can be used by attackers to violate the integrity or confidentiality of variables protected by the field blackbox, as the locations of their values in memory has changed. The native blackbox mode of operation (see (b)) provides more options for protecting sensitive values, as the blackbox is not located on the Java heap, but in an area of the JVM's process that is not actively maintained by the Java runtime. In both modes of operation, illegal access to sensitive values is hard. Their location in memory is unknown to the attacker, which prevents low-level attacks involving type confusion and buffer overflows. Further, reflective access is restricted and only possible through call sites that were deemed secure.

The transformation process is fast and completes within a few seconds. It would be possible to individualize every single JRE installation during setup, or even before every single application run. While this design is adequate for a proof of concept, we propose a more fundamental redesign of the platform for productive use, which we discuss in detail in Section 4.5.

4.3.4 Limitations

The lightweight solution presented here significantly strengthens Java's information hiding, as it addresses all attack vectors we previously identified. However, also this solution is not provably unbreakable. In the following, we will hence elaborate on residual risks and also discuss potential mitigation strategies.

One major concern of the lightweight solution is the use of secret tokens that are hardwired into the method bodies of system classes. The purpose of these tokens is to prevent untrusted code from accessing methods of the blackbox. While it is true that the Java runtime provides no means for attackers to inspect these method bodies, there are still potential attack vectors that may provide access to these tokens. This includes memory safety problems, such as format string errors,[7] or buffer over-reads.[8] The use of secret tokens in our proof of concept is the result of a deliberate design choice: it is a tradeoff that allows us to integrate our lightweight solution into closed-source Java runtimes without having to modify native components, at the cost of reduced attack resistance. The heavyweight solution we propose for productive use in Section 4.5 requires no secret tokens, but, as we will explain, its implementation requires modifications to the JVM.

Other concerns, besides the use of secret tokens, relate to the storage area of the blackbox, that is supposed to securely contain the contents of security-critical variables. Similar to the secret tokens in method bodies, also these values can potentially be leaked to attackers due to memory safety problems, if not protected properly. One mitigation strategy is to leverage software-based protection mechanisms, including, for example, process isolation, or address space layout randomization. Using a programming language like Rust [64] to implement the native blackbox components would provide strong guarantees with respect to memory safety, thus reducing the risk of introducing security-critical vulnerabilities. Additional mitigation strategies involve the use of dedicated hardware support. Trusted platform modules can be used to store sensitive values, which would significantly strengthen the implementation of the blackbox. Alternatively, as we explained previously in Section 4.3.2, the CHERI ISA [109], or similar solutions can be used to prevent untrusted code from accessing sensitive memory locations.

4.4 Evaluation

We next explain how we used our "lightweight" prototype to evaluate the following research questions:

RQ1: How effective is the blackbox in blocking existing attack vectors?
RQ2: To what extent does the blackbox retain backward compatibility with legacy applications?
RQ3: What is the performance impact the blackbox induces when executing real-world applications?

[7] see "CWE-134: Use of Externally-Controlled Format String"
[8] see "CWE-126: Buffer Over-read"

Table 4.2: Exploits that are blocked by the "lightweight" solution

Exploit	Countermeasures				Attack vector[1]	Attack category[2]
	Field blackb.	Integrity checker	Method blackb.	Classp. layout rand.		
Oracle JRE on Windows:						
IntegrityTestExploit	✓	✓	-	-	Introspec.	Multi-step
NO-CVE-27	✓	-	-	✓	Type conf.	Single-step
MULTI-CVE-2012-5075-2012-4681	✓	-	-	-	Introspec.	Restr. class
MULTI-CVE-2012-4681-2012-5074	✓	-	-	-	Introspec.	Restr. class
MULTI-CVE-2012-1682-2012-1726	✓	-	-	-	Introspec.	Restr. class
MULTI-CVE-2012-0547-2012-1726	✓	-	-	-	Introspec.	Restr. class
CVE-2013-2465	✓	-	-	-	Buff. overf.	Single-step
CVE-2013-1475	✓	-	-	✓	Type conf.	Single-step
CVE-2012-1726	✓	-	✓	-	Introspec.	Restr. class
CVE-2012-5076b	✓	-	-	-	Introspec.	Restr. class
CVE-2012-5076c	✓	-	-	-	Introspec.	Restr. class
CVE-2012-5076d	✓	-	-	-	Introspec.	Restr. class
CVE-2012-5076e	✓	-	-	-	Introspec.	Restr. class
CVE-2012-5076f	✓	-	-	-	Introspec.	Restr. class
CVE-2012-5076g	✓	-	-	-	Introspec.	Restr. class
CVE-2013-2423	✓	-	-	-	Type conf.	Single-step
CVE-2012-4681	✓	-	-	-	Introspec.	Multi-step
CVE-2012-5076a1	✓	-	-	-	Introspec.	Restr. class
IBM JDK on Ubuntu:						
NO-CVE-3	✓	-	-	-	Buff. overf.	Single-step
NO-CVE-8-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-9-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-10-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-11-ibm	✓	-	-	-	Introspec.	Multi-step
NO-CVE-12-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-13-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-14-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-18-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-19-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-20-ibm	✓	-	-	-	Introspec.	Restr. class
NO-CVE-24-ibm	✓	-	-	✓	Introspec.	Single-step
NO-CVE-26-ibm	-	✓	-	-	Introspec.	Single-step
NO-CVE-30-ibm	-	✓	-	-	Introspec.	Single-step

[1] According Section 4.2.2
[2] According Section 3.4.2

For purpose of this evaluation, we applied our prototype to original versions of Oracle JRE 7, and IBM JDK 7. We deliberately chose these versions because they best support the sample sets of exploits and real-world applications that we use in the following experiments.

4.4.1 RQ1: Effectiveness

We evaluated the effectiveness of the blackbox using the sample set of exploits presented in Chapter 3. This sample set was originally collected to cover a broad range of different attacks on the Java platform, including attacks that do not involve a breach of information hiding. Specifically for this evaluation, we added an additional exploits that we forked from one of the 61 minimal exploits to demonstrate that violating the integrity of a field that is accessed directly by the JVM is sufficient to compromise the Oracle JRE. The full sample set used in the experiments thus comprises 62 exploits. We set up a test environment to run the exploits on original and modified versions of Java that we created with our proof-of-concept implementation.

As a first step, we performed a manual review to evaluate how many exploits in the entire sample set actually break information hiding as part of the attack. This is important, as the original sample set comprises all attacks that we found in the wild while conducting the large-scale exploit analysis presented in Chapter 3, with no specific focus on information hiding or any other aspect of the Java security architecture. Several of the exploits in this set might thus be unrelated to information hiding, and should be removed from further consideration. As a result, we found that 38 our of 62 exploits use vulnerabilities to break information hiding. This is a larger share than we originally expected, considering that the original sample set was supposed to cover the broadest possible range of attacks on the Java platform. However, this result yet again highlights the importance of information hiding in the Java security architecture, and underlines the need for a systematic hardening.

As a second step, we evaluated how many of these 38 information-hiding attacks are blocked by our proof-of-concept solution. We found that our solution effectively blocks 32 exploits, which amounts to 84%. Table 4.2 provides details on the effectiveness of each of the four countermeasures. As can be seen, the field blackbox blocks 30 exploits (column 2), the integrity checker blocks three exploits (column 3), the method blackbox blocks a single exploit (column 4), and classpool layout randomization blocks three exploits (column 5). For Java, this is a first step in achieving defense in depth: considering that under normal conditions Java's regular protection of private class members is also in effect, all 32 exploits are now mitigated by at least two complementary countermeasures, five of them even by three.

Column 6 specifies the attack vectors implemented by the exploits according Section 4.2.2, which shows that the majority of attacks use vulnerabilities involving class introspection to break information hiding. This is not surprising, considering that in many cases these attacks are easier to implement and more reliable than type confusion attacks and buffer overflow attacks. Further, in column 7 we show the attack category as specified in Section 3.4.2. From this it can be seen that all type confusion and buffer overflow attacks were categorized as single-step attacks in Section 3.4.2, which means that implementing these attacks required just a single attacker primitive, i.e., one security vulnerability. Also, it can be seen that all multi-step attacks are blocked by the countermeasures we propose. All in all, this illustrates that our proof-of-concept

solution is capable of systematically blocking information-hiding attacks irrespective of whether the exploit was implemented using one single vulnerability or several different ones.

However, the countermeasures we propose were not able to block all information-hiding attacks in the sample set. Hence, finally, we reviewed the remaining six attacks on information hiding that are not blocked by our solution, and found that they implement the same attack vector, but using different security vulnerabilities as a foundation. All these exploits use defects that provide access to `ClassLoader.defineClass()`, a protected method that must be inaccessible to untrusted code, because it can be used to define high-privileged, custom classes. The method blackbox is not applied to this method to remain backward compatible—when there is no security manager in place, there may be user-provided subclasses of `ClassLoader` that call `defineClass()` for legitimate purposes. While our current prototype does not block these attacks, there are two possible solutions to block also this remaining vector.

The first solution is to simply apply the method blackbox also to `defineClass()` in cases when a security manager is put in charge. In original Java runtimes, untrusted code is in fact not allowed to load custom class loaders when a security manager is set, due to the dangers involving `defineClass()`. Instantiating a custom class loader is prevented by a permission check, but as six exploits in our sample set show, attackers find ways to bypass this restriction. The method blackbox could be used as an additional layer of defense to prevent illegitimate calls to `defineClass()` when a security manager is in effect, which would still retain backward compatibility, as original Java versions already prevent the instantiation of user-provided class loaders in this setting.

The second solution is to apply the "privilege escalation rule" proposed by Coker et al.[22], which prevents classes from loading other classes with higher privileges. It was specifically designed to mitigate attacks involving `defineClass()` or similar functionality. Although this countermeasure does not address information-hiding attacks in general, it is an ideal complement to our solution.

Based on the discussion above, we conclude that blocking all 38 attacks on information hiding in our sample set is possible.

4.4.2 RQ2: Backward compatibility

To evaluate the backward compatibility of the blackbox, we executed a small set of simple test applications, as well as all benchmarks in the DaCapo benchmark suite[12] version 9.12-bach on our modified versions of the Oracle JRE 7 and IBM JDK 7. The DaCapo suite consists of 14 different benchmarks, which represent complex real-world applications of various application domains, running non-trivial workloads. All applications completed as expected and we did not discover any misbehavior that resulted from our modifications. The countermeasures we propose are highly backward compatible. Integrating the blackbox into the JCL does not modify any public interfaces, the officially supported functionality of the platform is retained, and legacy applications run unaffectedly without changing configurations. As explained in Section 4.3, we provide special support for system classes that legitimately access private fields of other system classes, provided that the access is implemented in a secure, statically confined manner. This

Table 4.3: Results of our performance measurements. Absolute overheads are provided in milliseconds.

	Java blackbox				Native blackbox			
	w/o SM		w/ SM		w/o SM		w/ SM	
Benchmark	**abs.**	**rel.**	**abs.**	**rel.**	**abs.**	**rel.**	**abs.**	**rel.**
avrora	−23	−1%	−4	0%	−4	0%	−16	−1%
batik	−2	0%	-*	-*	1	0%	-*	-*
eclipse	437	3%	1925	10%	138	1%	2412	13%
fop	−5	−3%	−5	−2%	2	1%	−1	−1%
h2	−90	−2%	−79	−2%	8	0%	143	4%
jython	19	1%	31	2%	24	2%	47	3%
luindex	3	0%	2	0%	−25	−3%	35	3%
lusearch	−4	0%	−22	−2%	−21	−2%	5	1%
pmd	32	2%	58	3%	75	4%	113	6%
sunflow	−6	0%	−41	−2%	−9	0%	−21	−1%
tomcat	−113	−1%	295	2%	−225	−2%	1019	5%
tradebeans	−15	0%	21	1%	8	0%	72	2%
tradesoap	445	2%	−119	0%	32	0%	−337	−1%
xalan	11	1%	2	0%	−5	−1%	−9	−1%
Geom. mean		0%		1%		0%		2%

*batik cannot be executed with a security manager

is required to retain backward compatibility, as there are few system classes that rely on such access.

Even though we do not expect any issues with common real-world applications, there is a theoretical breach of backward compatibility with applications that attempt to access for legitimate reasons sensitive private members of system classes, which are protected by the blackbox. Because applications are never subject to our modifications, any attempt to access these private members will fail. However, we consider this acceptable, as we cannot construct any common use cases that would require such access. Also, writing applications that directly access private members of system classes is generally not just considered bad style, but also insecure, because also those applications can have confused-deputy vulnerabilities in which they reexpose the sensitive functionality.

4.4.3 RQ3: Performance

To evaluate the performance impact of our proposed countermeasures we conducted another large-scale set of experiments. For this, we used the DaCapo benchmark suite again, because it was specifically designed for performance evaluations like these [12].

We executed all benchmarks under several configurations. First, we run all benchmarks using an unmodified Oracle JRE 7 installation, which serves as a baseline for comparisons. Then, we run all benchmarks on the modified Oracle JRE 7 that contains the blackbox mechanisms. This is performed twice per benchmark: once in the "Java blackbox" mode of operation, which stores sensitive values in a dedicated Java class, and once in the "native blackbox" mode of operation, which stores sensitive values in a native component. Finally, all tests are executed two times, once without a security manager, and once with the default security manager in place with a security policy that grants all permissions to all code. The execution of a single benchmarks under any configuration comprised 500 timed application runs. We excluded the first 100 runs, which served only as a warmup, and reported the average of the remaining runs as a result. Measurements were performed on Windows 10, Intel Core i5-3350P 3.10 Ghz CPU, with 12 GB RAM.

Table 4.3 shows the results of our performance tests. For each benchmark, we report the absolute ("abs.") and relative ("rel.") overheads induced by our changes. It is important to note that the total runtimes of the individual benchmarks vary greatly among each other, e.g., fop completes in much shorter time than tomcat.

As can be seen, in the "Java blackbox" mode of operation, all overheads lie in the range of -3% and 10%. Except for two outliers, overheads induced by the blackbox are between -2% and 3%. Only seven benchmarks show a slowdown at all. Negative overheads imply that our modifications increased execution speed, however, this is rather unlikely. Our measurements are subject to small inaccuracies, which are not caused by deficiencies in our experimental setup, but rather inherent to all Java benchmarks executed on a real hardware/software stack. Gil et al. [36] studied this effect in depth and found that even executing identical code can lead to slightly different runtime measurements. Gu et al.[46] identified various factors that can lead to variances in execution speed of Java applications.

The performance measurement results for the "native blackbox" mode of operation are similar, but we observe slightly higher overheads for some benchmarks. Overall, overheads lie between -3% and 13%. Excluding two outliers, overheads range from -2% to 6%. We expected these higher overheads, because the native blackbox involves a higher number of JNI calls, which are costly compared to regular Java calls.

The highest overheads in both modes of operation can be seen for Eclipse. The reason is that this benchmark in the DaCapo suite is itself a collection of performance testes for the Eclipse Java Development Tools, which frequently execute a specific call sequence that became slower through our modifications. Even though Eclipse is a real-world application, the workload it performs in the DaCapo benchmark suite does not at all resemble typical user behavior.

In summary, integrating the blackbox into the Oracle JRE 7 resulted in average overheads, i.e., geometric means, between 0% and 2%, depending on the mode of operation, and the status of the security manager.

4.5 Solution for productive use

The proof of concept we presented in Section 4.3 illustrates the importance of information hiding in Java platform security, and that there can be a hardening that blocks a broad range of attack

vectors, while remaining backward compatible and high-performance. The blackbox mechanisms were shaped by constraints and assumptions that allowed for large-scale experiments on multiple platforms, while keeping implementation efforts at a reasonable cost. The following presents ideas on how to strengthen information hiding for productive use, overcoming weaknesses of our proof-of-concept approach.

The first step in improving the security design of the platform would be to consolidate the core of policy enforcement in a central, isolated component, which we call the *Java security monitor*. This includes all data structures that determine the security status of the system (e.g., whether security mechanisms are enabled or not), data structures involved in policy enforcement (security policies, protection domain descriptions, etc.), and all algorithms that operate on these data structures (e.g., access control algorithm). The only way to access the security monitor must be through dedicated, well-defined, public interfaces. Access to its inner workings through direct memory access, reflection, etc., must be prevented by all means. One way to implement such a component would be in native code, using state-of-the-art countermeasures to prevent tampering, possibly including stack canaries, control-flow integrity checks, ASLR, and so forth. JNI can be used to implement public interfaces for permission checks and other security-related functionality. The security monitor must only comprise the very core functionality needed to implement policy enforcement, so that it remains as small and simply as possible.

For protecting sensitive members in system classes that do not belong to the core of policy enforcement (like `Statement.acc`), we propose to extend the Java Language Specification [80] by an additional modifier `critical`. Members modified this way should be protected in the same way as the blackbox protects fields: reflective access should, if at all, only be allowed when performed in a statically-confined manner, and the member's location in memory should be randomized to prevent low-level attacks. Further, we propose to extend the JVM instruction set by a new set of instructions whose use is reserved for system classes only, see Appendix A for a complete overview. These instructions, on the one hand, comprise dedicated commands that system classes must use to call private methods. On the other hand, the new instruction set comprises dedicated commands for `critical` field access. Since application classes can not use these instructions, they are systematically prevented from accessing sensitive members directly, or by means of reflection. This design prevents illegitimate access to sensitive members without the use of secret tokens, which reduces the attack surface compared to the proof-of-concept implementation presented in Section 4.3.

Specifically, we propose the following strategy to implement our proposed changes:

1. Consolidate the core of policy enforcement in a natively implemented security monitor, which includes all functionality required to carry out permission checks. The security monitor must further comprise a dedicated, secure data store for `critical` field values, like `Statement.acc`. The data store's structure and location in memory must be randomized, and the security monitor's implementation must employ state-of-the-art mitigation strategies to prevent buffer overflows and code-reuse attacks—using a language like Rust [64] that provides strong memory safety guarantees by design is possible.
   The JVM must provide two modes of operation. In the first mode, the security monitor is linked to the JVM's binary and both components operate in the context of the same process. This allows for high performance, but provides little isolation.

The second mode of operation executes the security monitor in a dedicated process, which leverages the operating system's capability of process separation to further increase attack resistance, however, at the cost of depending on costly inter-process communication.

2. Extend the JVM's bytecode interpreter and JIT compiler by the new set of instructions that we propose for sensitive member access. Using these instructions allows system classes to call private methods, and to load and store sensitive values in the security monitor's secure data store, which do not belong to the very core of policy enforcement, like `Statement.acc`.
3. Modify the platform's bytecode verifier to provide support for the new set of instructions. This includes, on the one hand, a basic understanding that the new instructions are valid commands that may be part of a method body. On the other hand, the bytecode verifier, as a first level of defense, must ensure that only system classes make use of these instructions. As there is no self-modifying code in Java, this effectively and systematically prevents application code from accessing the contents of the security monitor's data store, or calling private methods in system classes. Taken together, the extension of the JVM instruction set and this modification of the bytecode verifier eliminate the need for secret tokens, which is a major difference to the proof of concept that we presented in Section 4.3.
4. Modify the Java compiler as follows:
   (a) System classes that access a `critical` field must use the extended instruction set for this to ensure that values are stored and loaded from the security monitor's secure data store.
   (b) System classes must use the extended instruction set to call private methods.
   (c) Extend the compiler by a static analysis that detects reflective access from a system class to `critical` members of other system classes, using a similar approach like our proof-of-concept implementation. If the analysis finds a statically-confined, hence secure, access to a private member, the compiler must treat this like a regular, legal `critical` member access and use the new instruction set to allow the access. Reflective access that is not statically-confined, hence insecure, must be kept unchanged to prevent confused deputy attacks. Our proof of concept shows that this strategy successfully retains backward compatibility, while blocking attacks at the same time.
5. Extend the runtime to enforce the "privilege escalation rule" proposed by Coker et al.[22], which prevents classes from loading other classes with higher privileges. This specifically guards against exploits that define high-privileged classes to achieve arbitrary code execution, e.g., by abusing `ClassLoader.defineClass()`, or similar functionality. Although this countermeasure alone is by no means sufficient to address information-hiding attacks in general, it ideally complements the countermeasures that we propose to harden the platform.

Despite the conceptual simplicity and robustness of the proposed solution, the above list of code changes illustrates that significant engineering effort is required to implement the solution for productive use. However, the solution we presented in Section 4.3 highlights the importance of the subject and the need for a systematic hardening of Java's information hiding, which justifies even major implementation efforts. Although we propose various changes to the internal workings of the platform, all interfaces that application code can use to interact with system classes remain

unchanged—from the application's point of view, even our solution for productive use is fully backward compatible. Consequently, all application classes can be executed on the new platform without any changes of reconfigurations.

While the above solution strengthens the Java platform's security in general, and information hiding in particular, it also overcomes two weaknesses of our proof-of-concept solution. First, the solution outlined here does not depend on any secret tokens hardwired into the method bodies of system classes, which further reduces the attack surface. Second, implementing this solution for productive use presumably induces little to no overheads. Slowdowns we observed with our proof of concept were primarily caused by a large number of JNI calls. The solution proposed here, however, is consolidated and integrated into the native components of the platform, which requires much fewer costly JNI calls between the JCL and JVM.

4.6 Related work

Various different approaches to strengthen the Java platform have been proposed in the past. However, as we explain in the following, none of them comprehensively mitigate common attacks on information hiding.

Java-specific attack mitigation strategies

The Java platform features a countermeasure that specifically addresses the dangers of insecure use of reflection: so-called *filter maps*. Any Java class can use public interfaces of the reflection API to add individual class members to filter maps, which hides them from reflective access, similar to a blacklist. The field `System.security` is an example of a field that is added to the field filter map by default—calling `System.getDeclaredFields()` will thus return all declared fields except for `security`. Listing 4.4 illustrates how this is implemented in Java 8. As can be seen, `Class.getDeclaredFields()` calls a private helper method `Class.privateGetDeclaredFields()` in line 6, which in turn calls `Reflection.filterFields()` in line 11. This method is responsible for filtering out all fields that were previously added to the field filter map, see line 31, using another helper method `filter()` not shown in this listing. This countermeasure is generally ineffective against low-level attacks involving, e.g., type confusion, as sensitive field values remain in the same memory locations. Filter maps could, in theory, be effective against attacks involving confused deputies that insecurely use reflection, but in practice, many sensitive fields are not added to filter maps.

Coker et al. [22] proposed two countermeasures that seek to block a variety of different attacks without breaking backward compatibility. The "privilege escalation rule" prevents classes from loading other classes with higher privileges, the "security manager rule" enforces that a security manager, once it has been set, cannot be changed or disabled. In their evaluation, Coker et al. use a sample set of ten exploits to show that the privilege escalation rule blocks four attacks, and the security manager rule blocks all of them. We reviewed the exploits in this sample set and found that the privilege escalation rule does not comprehensively protect against certain attacks on information hiding. For example, the exploit for CVE-2013-2423, which was used by

Listing 4.4: Implementation of field filter maps in Java 8

```
1  //java.lang.Class
2  public final class Class<T> implements java.io.Serializable,
       GenericDeclaration, Type, AnnotatedElement {
3    // ...
4    public Field[] getDeclaredFields() throws SecurityException {
5      // ...
6      return copyFields(privateGetDeclaredFields(false));
7    }
8    // ...
9    private Field[] privateGetDeclaredFields(boolean publicOnly) {
10     // ...
11     res=Reflection.filterFields(this,getDeclaredFields0(publicOnly));
12     // ...
13     return res;
14   }
15   // ...
16 }
17
18 //sun.reflect.Reflection
19 public class Reflection {
20   // ...
21   static {
22     Map<Class<?>,String[] map=new HashMap<Class<?>,String[]>();
23     // ...
24     map.put(System.class, new String[] "security");
25     fieldFilterMap=map;
26     // ...
27   }
28   // ...
29   public static Field[] filterFields(Class<?> containingClass,
         Field[] fields) {
30     // ...
31     return (Field[])filter(fields,fieldFilterMap.get(containingClass));

32   }
33 }
```

Coker et al. for evaluation and was not blocked by this rule, implements a common attack on information hiding using type confusion to get access to sensitive fields. The ineffectiveness of this rule in that case is expected, because the attack does not involve the loading of classes with higher privileges. The countermeasures we propose in this chapter, in contrast, effectively block this attack.

However, the proof-of-concept we implemented was unable to mitigate attacks that used `ClassLoader.defineClass()` to define and load custom, high-privileged classes, see Section 4.4.1. For this reason, as we explain in Section 4.4.1 and 4.5, the privilege escalation rule is an ideal complement to the countermeasures we propose, as it was specifically designed to block attacks involving `ClassLoader.defineClass()` or similar methods.

The results for the security manager rule seem promising at first, as all ten exploits in the sample set used for evaluation are blocked. However, this raises the question of *recoverability*: How resistant is this countermeasure against modifications of the original exploits? Due to the in-depth study of a broad range of Java exploits presented in Chapter 3 we know that there is a high potential for recovering exploits that were originally blocked by the security manager rule. The reason for this is that in many attack vectors the deactivation of the security manager is not a necessary prerequisite for the attack to be successful, but rather a matter of convenience—once arbitrary code execution has been achieved through other means, exploits use their newly gained privileges to disable the security manager, which ensures that a payload can be executed without any specific constraints or restrictions. However, this is not a necessary step, as arbitrary code execution has already been achieved, any payload can be executed successfully even with the security manager still in place. The security manager rule is thus not very resistant against modified attacks, it rather breaks backward compatibility with existing exploits.

Alternatives to Java's approach to stack inspection

Several alternatives to Java's approach to stack-based access control were proposed in academia. Abadi and Fournet [5] presented history-based access control. In contrast to Java's original implementation, their approach not only considers the methods that are currently on the call stack when a permission check is triggered, but also all methods that already completed execution in the past. Pistoia et al. [89] presented a variation of this approach that considers only the subset of methods that are actually involved in a security-sensitive operation, which increases precision. While these proposed solutions overcome certain shortcomings of Java's stack inspection routine, they were not designed to strengthen information hiding. Especially low-level attacks involving type confusion and buffer overflows are not in scope of these mechanisms, consequently, they do not provide sufficient protection against these attacks.

Detection and prevention of buffer overflows

Various techniques have been proposed to detect and prevent buffer overflows in native code [25, 26, 95, 61], including approaches to enforce control-flow integrity [4, 113]. Although this field of research advanced in the past years, buffer overflows are still considered a challenge today, as many approaches suffer from practical problems, like high performance overheads, or low precision. Moreover, none of these techniques address illegal field access in Java through type confusion, or by means of a confused deputy.

Classpool layout randomization is a countermeasure that achieves randomized memory locations of sensitive fields of Java classes. Different variants of address obfuscation and randomization have been proposed and implemented in the past [101, 57, 11], however, to the best of our knowledge, we are the first ones to implement this specifically for Java class files.

Hardware-based memory protection
Various different approaches to enforce memory protection with hardware support were proposed in the past. Most relevant to the work presented in this chapter is the CHERI Instruction-Set Architecture (ISA) [109] which implements fine-grained, capability-based memory protection at the hardware layer. Our prototype implementation of the blackbox uses software-based mechanisms and secret tokens to prevent illegal access to the contents of the blackbox. As we explain in Section 4.3.2, the CHERI ISA could be used to separate the contents of the blackbox and restrict access to only legitimate callers on the hardware level, thus further strengthening the implementation of our proposed concept. Chisnall et al. [17] used the CHERI ISA to implement a sandboxing mechanism for native code in Java, which ensures that also this code adheres to Java's security and memory model, thus preventing access to security-critical memory locations.

Redesigning Java's security architecture
In Section 4.5 we propose a fundamental redesign of the Java platform's security mechanisms in order to systematically mitigate information-hiding attacks, which would potentially increase the Java runtime's overall security. In the past, other approaches to redesign Java's security architecture were proposed in academia.

Toledo et al. [106] found that security-related code in the JCL is scattered all over the codebase, and they show how aspect-oriented programming can be used to modularize access control. Their proposed solution increases maintainability and allows for finer-grained policies, but low-level attacks on information hiding are not covered by this.

Ion et al. [55] extended the security architecture of the J2ME, an edition of the Java platform specifically for mobile devices, in order to support the enforcement of fine-grained security policies. Similar to our proof-of-concept implementation, they designed their prototype such that it can be integrated into the runtime environment without requiring modifications to the JVM. However, strengthening information hiding is not in scope of this work.

4.7 Conclusion

We studied the role of information hiding in the Java security architecture and showed that a large portion of the exploits we collected and presented in Chapter 3 depend on illegal access to private members of system classes in order to implement their attacks.

To address this problem, we present two different approaches to systematically strengthen the Java platform against common attacks on information hiding. The *lightweight* mitigation strategy combines a set of four countermeasures that our proof-of-concept implementation can integrate into even closed-source JREs. An evaluation shows that this solution blocks 84% of all information-hiding attacks in our exploit sample set, and we explain how even the remaining

attacks can be blocked as well. We further show that this solution retains backward compatibility and high performance—for a sample set of complex real-world applications, the runtime overhead induced by our changes lies below 2%.

The alternative, *heavyweight* mitigation strategy that we propose suggests a fundamental redesign of the internals of the Java runtime. Specifically, we propose to consolidate the core of policy enforcement into a central, isolated component, the Java security monitor, which also features a dedicated, secure data store for sensitive field values. Further, we suggest extensions to the Java Language Specification and Java Virtual Machine Specification to prevent illegal access to private members of system classes, and ensure that sensitive values will be stored in the security monitor's data store. As we explain, implementing this mitigation strategy would require major engineering efforts, but it has the potential to outperform our lightweight proof of concept in terms of both robustness and speed.

5

Hardening Java's access control[1]

The previous chapter presented a solution to strengthen Java's information hiding which hardened the platform against a large number of known attacks. The primary goal of this effort was to increase the Java platform's attack resistance. In this chapter, we will take a different perspective and put a focus on security-code maintainability. Specifically, we will show that under certain circumstances the Java runtime will skip permission checks based on heuristics that are hardcoded into the methods of system classes in order to improve performance. We will argue that these shortcuts significantly decrease maintainability, even to the point that they enabled the introduction of security vulnerabilities in the past. Moreover, we will present a solution to remove these shortcuts and show that with current technology this change does not impact the performance of a set of real-world applications. We further discuss considerations for the productive use of our proposed solution, present ideas for further research, and conclude with lessons learned that provide guidance on the design and implementation of secure software beyond our original scope.

5.1 Motivation and contributions

The Java runtime was designed to execute code originally received from potentially untrusted sources. Security was thus a major design goal for the platform to ensure that the host machine that executes the runtime is protected from harm. As explained in Section 2.2.1 a fundamental concept in Java's security architecture is access control implemented by means of stack-based permission checks. Sensitive methods in the JCL are guarded by a call to the security manager that triggers a security check, making sure that all callers in the access control context were assigned sufficient privileges for the attempted action, such as local file or network access.

Java's security model centers around these permissions and its flexibility allows administrators and runtime users to supply their own security policies which are then enforced dynamically by the runtime. At the same time, the concept of stack-based permission checks to implement access control in Java largely prevents by design that JCL developers accidentally introduce privilege escalation vulnerabilities, unless they explicitly elevate privileges using the privileged block API, see Section 2.2.1.

[1] Parts of this chapter were taken directly or with modifications from [51].

However, as we find, this is only theory. In practice, it shows that many security-sensitive methods in the JCL implement what we call *shortcuts*: They execute permission checks only under certain circumstances and use heuristics, such as checking the type of only the immediate caller's class loader, to validate the secure execution in other cases. Methods with shortcuts are generally caller sensitive: Depending on the nature of the shortcut, they grant privileges implicitly to certain groups of callers, in many cases to all callers within the JCL. As such, they are a counterpart to the privileged block API. One significant difference between the two is that privileged blocks are always *explicit* and shortcuts elevate privileges *implicitly*. As we will show in this chapter, this difference has major consequences on the security and maintainability of the platform.

The extensive exploit analysis we presented in Chapter 3 showed that, besides weak information hiding, improper access control is a major design flaw in Java's security architecture. In particular, attackers can abuse insecure use of reflection in system classes to invoke shortcut-containing methods, which help them break Java's security guarantees. More generally, we showed that caller sensitivity in combination with confused deputies is a primary attack vector for exploits in the wild, which highlights the risk of caller sensitivity to the Java platform.

Moreover, we will show that shortcuts severely impede the maintainability of the Java runtime's implementation. During our investigation we found several places inside the runtime library at which developers could inadvertently break the entire runtime's security architecture through simple code transformations that would otherwise be considered semantics-preserving refactorings. In particular this is true for the introduction of wrapper methods. Wrappers modify the call stack, which can inadvertently cause the shortcuts to check properties of the wrong stack frames. Incidentally, these places contained code comments with explicit warnings to prevent the introduction of vulnerabilities, such as "This function performs stack walk magic: do not refactor it".[2] In addition to cases, where a shortcut poses an immediate threat, there are often cases in which it can lead to an exploitable vulnerability later, due to simple code maintenance and evolution.

We conducted an experiment, transforming a JCL release such that explicit privileged blocks become the *only* way in which the JCL elevates privileges. This has several advantages. First, as we elaborate later, it eliminates certain attack vectors that abuse insecure use of reflection to profit from shortcuts. Second, it makes privilege elevation *explicit*, which eliminates the potential to introduce privilege escalation vulnerabilities accidentally through code restructuring and evolution. Third, explicit privilege elevation allows both security experts and code analysis tools to focus on privileged blocks to ensure the security of the access-control implementation.

One prevalent reason for introducing shortcuts in the first place is that stack-based permission checks are expensive. After all, the JVM needs to reify the call stack and check permissions for all callers involved in the action. Shortcuts presumably lead to a faster implementation of access control. However, in this work we show through a set of large-scale experiments that no such penalty is measurable on the DaCapo benchmark suite, despite the fact that it makes heavy use of security-sensitive APIs, and also state reasons for why this is the case.

[2] `http://hg.openjdk.java.net/jdk7u/jdk7u-dev/jdk/file/9a5aa33fadc4/src/share/classes/java/lang/invoke/MethodHandles.java`

A second reason for the presence of shortcuts is that the implicit assignment of privilege is convenient, as it reduces the need to elevate privileges explicitly, e.g., through an appropriate access-control policy. Another contribution presented in this chapter is thus a detailed assessment of the usability implications that a move from implicit to purely explicit privilege elevation entails. This assessment allows us to provide specific guidance for an actual implementation of our hardening in Java's codebase. Last but not least we discuss lessons learned that shall provide guidance on the design of secure software beyond the scope of Java.

In summary, this chapter presents the following contributions:

- the first detailed analysis of the effects of implicit privilege elevation and shortcuts for access-control checks in Java, along with the security and maintainability problems they induce,
- a tool-assisted analysis and adaptation technique to avoid the risk of introducing confused deputies in the JCL due to shortcuts,
- an adapted version of the JCL that implements access control without shortcuts, a detailed explanation of why this adapted version enhances security and maintainability,
- a set of large-scale experiments showing that this added security and maintainability comes at a negligible runtime cost, and
- guidance on the productive use of our proposed solution, and an outline of open research questions as well as general lessons learned from our in-depth analysis.

All artifacts needed to reproduce our results are publicly available.[3]

5.2 Comparison of privileged blocks and shortcuts

Section 2.2.1 explains how the Java runtime implements access control by means of stack-based permission checks to prevent untrusted code from accessing sensitive APIs and system resources. As stated above, the privileged block API and shortcuts are two different concepts that both extend the basic approach of stack inspection and influence how the Java runtime checks the authorization of callers that are involved in a security-sensitive action. More specifically, they can both be used to elevate the privileges of a subset of callers on the stack to allow that certain well-defined, sensitive actions in system classes can also be triggered by untrusted code, which would otherwise fail due to a lack of permissions. Besides these commonalities between privileged blocks and shortcuts, however, there are substantial differences between the two concepts. Before subsequent sections discuss their advantages and disadvantages, we will first illustrate in more detail how the concepts work and how they are different.

5.2.1 Privileged blocks

The privileged block API is a feature of the Java runtime that allows code with appropriate permissions to explicitly elevate privileges for specific operations by a call to `doPrivileged()`. Section 2.2.1 explains in more detail how security-sensitive operations can be contained in a privileged action, which enables trusted code to act as a guarantor that performs a certain action

[3] `https://github.com/stg-tud/jdeopt`

Listing 5.1: Example of a proper permission check

```
1   class FileAccess {
2     File openFile(String path) {
3       SecurityManager sm = System.getSecurityManager();
4       sm.checkPermission(new FilePermission(path));
5       return newFileHandle(path);
6     }
7   }
8
9   class SystemProperties {
10    public String readProp(String name) {
11      File file = AccessController.doPrivileged(
12      (PrivilegedAction<File>) () -> FileAccess.openFile(JDK_PATH
            + "/system.properties"));
13      ... // read property
14    }
15  }
16
17  // code below this point is added in a later release
18  class Util {
19    public File openFileFromRoot(String name) {
20      // will throw SecurityException
21      return FileAccess.openFile("/" + name);
22    }
23  }
```

on behalf of untrusted callers that would otherwise be prevented from performing the action themselves. Naturally, the trusted code that elevates its privileges to execute the action must ensure that all security-sensitive operations performed in that context cannot cause harm even if triggered by malicious code.

For illustration, consider Listing 5.1. In this example, `SystemProperties.readProp()` uses `AccessController.doPrivileged()` in line 11 to temporarily elevate the privileges of the executing thread such that the call to `SecurityManager.checkPermission()` in `FileAccess.openFile()` in line 4 can succeed even in cases where the public method `SystemProperties.readProp()` is called from untrusted code. This is intended behavior and a demonstration of how a trusted class, `SystemProperties`, explicitly acts as a guarantor. In the example, the "privileged" call to `FileAccess.openFile()` in line 12 is explicitly trusted to not cause any harm irrespective of the callers of `SystemProperties.readProp()`, in this case rightfully so, as the constant arguments ensure that only the system properties file will be accessed without exposing a sensitive file handle to a potential attacker.

Listing 5.2: Example of a shortcut

```
1  class FileAccess {
2    File openFile(String path) {
3      // if caller is not a trusted system class
4      Class caller = Reflection.getCallerClass();
5      if(caller != null && ClassLoader.getClassLoader(caller) != null) {
6        SecurityManager sm = System.getSecurityManager();
7        sm.checkPermission(new FilePermission(path));
8      }
9      return newFileHandle(path);
10   }
11 }
12
13 class SystemProperties {
14   public String readProp(String name) {
15     File file = FileAccess.openFile(JDK_PATH + "/system.properties"
           );
16     ... // read property
17   }
18 }
19
20 // code below this point is added in a later release
21 class Util {
22   public File openFileFromRoot(String name) {
23     return FileAccess.openFile("/" + name);
24   }
25 }
```

5.2.2 Shortcuts

We say that a method contains a *shortcut* if it contains a permission check, i.e., a call to a method of the form `SecurityManager.check*()`, that is carried out only if certain constraints on the current call stack are unsatisfied. These constraints are expressed through conditionals and typically take the immediate caller and/or its class loader into account. The implementation of shortcuts is not supported by a dedicated Java language construct or API.

`Class.getDeclaredMethods()` is an example for a method that implements a shortcut. It skips a permission check in case the immediate caller was defined by the same class loader as the class whose members shall be accessed by the call. The Secure Coding Guidelines for Java [82] list a number of such caller-sensitive methods in Sections 9.8 through 9.11 that should be used with special care to avoid the introduction of vulnerabilities.

For illustration, consider the simplified example in Listing 5.2. Assume that classes `FileAccess` and `SystemProperties` exist in some release of the JCL and class `Util` has been introduced in a later release to the trusted class library, as a convenience. The method `FileAccess.openFile()` opens arbitrary files on the caller's behalf. Since this is a security-sensitive operation, the method checks for the appropriate `FilePermission` in line 7. However, this check

is not performed for all callers, as the method implements a shortcut: A conditional in line 5 checks the type of the immediate caller's class loader, and skips the permission check if the immediate caller is associated with a `null` class loader. As previously explained in more detail in Section 2.1.4, all classes in the JCL, including `FileAccess` and `SystemProperties` here, are associated with the class loader `null`, as this represents the bootstrap class loader that loads all system classes. From this it follows that, by taking the shortcut, the method `openFile()` *implicitly trusts* all callers that reside in the JCL.

This is no problem with callers that, knowingly or unknowingly, use this implicitly gained privilege carefully. For instance, the method `System.readProp()` exhibits no vulnerability, as it only reads the system properties file it needs, similar to the code in Listing 5.1. However, it is fairly easy to accidentally expose the elevated privilege to untrusted callers, which in many cases may result in a privilege escalation vulnerability that attackers can exploit to cause harm.

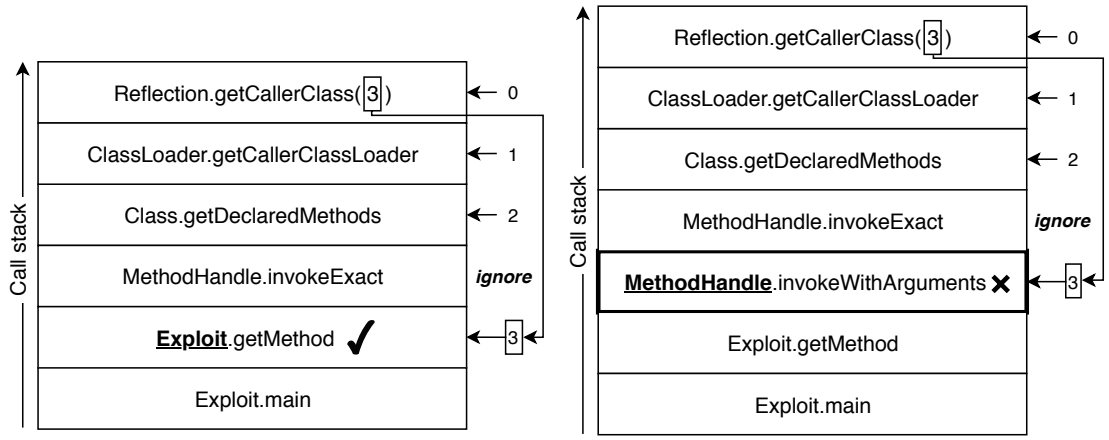
For example, to the developer of the new class `Util` in Listing 5.2, it is by no means obvious that the introduction of a simple wrapper method could have severe security implications: The method `Util.openFileFromRoot()` is an example of a confused deputy, as it exposes the complete behavior of `FileAccess.openFile()` to its callers, without any filtering, checking, or sanitization of the passed arguments. Consequently, any user-supplied code, including malicious applications and untrusted applets, can misuse `Util` to bypass all permission checks within `openFile()`, as `Util` is trusted, i.e., loaded by the bootstrap class loader and hence associated with a `null` class loader. As already discussed in Chapter 3, in the past, accidentally introduced confused deputies like `Util` have actually led to severe vulnerabilities in the JCL that allowed attackers to completely bypass all security mechanisms, thus achieving arbitrary code execution on the target machine. Note that in Listing 5.1, the introduction of this wrapper method has no negative side effects, as in this example, the method `FileAccess.openFile()` does not implement a shortcut.

5.3 Problem statement

The previous section already explained that shortcuts are problematic when introducing wrapper methods that change the call stack in such a way that properties of the wrong stack frames are checked, thus ultimately leading to potential security vulnerabilities. This section elaborates more on the problems and disadvantages of shortcuts in methods of trusted system classes. Specifically, the fact that shortcuts implicitly grant privileges to a certain subset of callers negatively impacts the Java runtime's attack resistance and code maintainability, as the following explains.

5.3.1 Increased attack surface

Shortcuts in the JCL increase the number of potential attack vectors. Attackers can abuse reflection or method handles to call shortcut-containing methods on behalf of a trusted class. Many of these methods will then skip a permission check, because the immediate caller is trusted, and thus provide functionality that was intended to be restricted. Two examples of such methods that are known to be of great value to attackers are `Class.getDeclaredFields()` and `Class.getDeclaredMethods()`, which skip permission checks, if the immediate caller



(a) No security breach

(b) Security breach caused by an incorrect check

Figure 5.1: Shortcut checks the wrong stack frame due to a wrapper method

is defined by the same class loader as the class whose members shall be accessed by the call. They can be used to access private members of a class that were intended to be inaccessible by untrusted code. To find specific examples of such kinds of attacks, we manually revisited the sample set of exploits we originally received from Security Explorations, see Chapter 3. We found that at least four of those original 52 exploits depend on shortcuts.[4]

One of these example exploits, issue 32, profits from a vulnerability that is filed under CVE-2012-5088. The underlying problem is that the implementation of shortcuts requires functionality that provides details about the call stack, such as type information of the individual callers on the stack. There are several methods in the JCL that offer such functionality. For example, `Reflection.getCallerClass(int index)`, which is also used to implement the shortcut in `Class.getDeclaredMethods()`, provides type information of the caller at the specified index.

At a first glance, it may seem trivial to implement methods like `getCallerClass()`. However, taking introspection APIs like Java's reflection API and method handles into account, this task is more complicated. The runtime is supposed to hide the fact that a method was called reflectively, to ensure that even caller-sensitive callee methods behave as if they were called immediately. To achieve this, methods like `getCallerClass()` do not report on the use of introspection APIs, thus entirely hiding any layers of indirection. For example, Figure 5.1a shows that, under normal circumstances, a shortcut-containing method like `getDeclaredMethods()` checks the right stack frame, even when being called through methods like `invokeExact()`, because `getCallerClass()` hides the layer of indirection. This is necessary, because in the context of `getDeclaredMethods()`, a constant index is provided to `getCallerClass(int index)`, which means that the call stack is always assumed to have the same layout, irrespective of how `getDeclaredMethods()` was called.

The vulnerability CVE-2012-5088 refers to a case in which the runtime selectively reported a method in `MethodHandle` as a caller, thus not properly hiding the layer of indirection caused by

[4]Issues 32, 35, 36, and 37

Listing 5.3: Demo code to illustrate four different ways of calling a method

```
1  import java.lang.invoke.MethodHandle;
2  import java.lang.invoke.MethodHandles;
3  import java.lang.invoke.MethodType;
4  import sun.reflect.Reflection;
5
6  public class Main {
7
8    public static void main(String[] args) throws Throwable {
9      MethodType desc = MethodType.methodType(Void.TYPE, new Class[]
           {String.class});
10     MethodHandle handle = MethodHandles.lookup().findStatic(Main.
           class, "printCallers", desc);
11     printCallers("Case 1: Direct call");
12     handle.invoke("Case 2: MethodHandle.invoke()");
13     handle.invokeExact("Case 3: MethodHandle.invokeExact()");
14     handle.invokeWithArguments("Case 4: MethodHandle.
           invokeWithArguments()");
15   }
16
17   public static void printCallers(String message) {
18     System.out.printf("===%s===\n", message);
19     StackTraceElement[] trace = new Exception().getStackTrace();
20     for(int i = 0; i < trace.length; i++) {
21       StackTraceElement e = trace[i];
22       String fullName = Reflection.getCallerClass(i+1).getName();
23       String simpleName = Reflection.getCallerClass(i+1).
             getSimpleName();
24       System.out.printf("%d: (%s) %s.%s:%d\n", i+1, fullName,
             simpleName, e.getMethodName(), e.getLineNumber());
25     }
26     System.out.print("\n");
27   }
28
29 }
```

Listing 5.4: Output of demo code

```
1  ===Case 1: Direct call===
2  1: (Main) Main.printCallers:19
3  2: (Main) Main.main:11
4
5  ===Case 2: MethodHandle.invoke()===
6  1: (Main) Main.printCallers:19
7  2: (Main) Main.main:12
8
9  ===Case 3: MethodHandle.invokeExact()===
10 1: (Main) Main.printCallers:19
11 2: (Main) Main.main:13
12
13 ===Case 4: MethodHandle.invokeWithArguments()===
14 1: (Main) Main.printCallers:19
15 2: (java.lang.invoke.MethodHandle) MethodHandle.invokeWithArguments
       :566
16 3: (Main) Main.main:14
```

the use of method handles. To illustrate this special case, consider the demo code in Listing 5.3. It shows four different ways of calling an example method `printCallers()`, which, as its name suggests, prints information about the call stack as reported by the runtime. Following the above explanations, one may assume that in all four cases the method reports the same set of callers. Listing 5.4 shows the output generated by the demo code when executed on Java 7. As can be seen, the set of callers reported by `printCallers()` is only identical in three out of four cases. In the fourth case, when calling a target method using `MethodHandle.invokeWithArguments()`, the runtime will report `invokeWithArguments()` as the immediate caller, see line 15. Interestingly, `invokeWithArguments()` serves as a wrapper method to `invokeExact()`, but `invokeExact()` is not reported as a caller. This misbehavior represents a real-world example of how the introduction of a new wrapper method can lead to an exploitable security vulnerability in a runtime that contains shortcuts.

Attackers can exploit this inconsistency by using `invokeWithArguments()` to call a shortcut-containing method. Figure 5.1b shows how the hardcoded index provided to `getCallerClass()` in the implementation of the shortcut in `getDeclaredMethods()` refers to the wrong stack frame in this case, thus providing access to otherwise restricted functionality.

The vulnerability we discussed here is just one example. We showed in Chapter 3 that caller sensitivity in combination with confused deputies is a very common attack vector used by attackers.

5.3.2 Decreased maintainability

Besides increasing the attack surface, shortcuts also significantly decrease the maintainability of the code of system classes. The potential is high that developers, who are either unaware of or unable to properly reason about the implicitly granted privileges, introduce security flaws when maintaining and extending the JCL by implementing new callers of methods with shortcuts. In the following, we provide a set of reasons for why this is the case.

Missing documentation
Information about shortcuts is rarely part of the method's documentation. Hence, developers of any caller methods will not be aware that calling certain methods imposes requirements on their implementation to not expose critical functionality to untrusted code. Consider again `Class.getDeclaredMethods()` and the scenario, where a maintainer of the JCL introduces a wrapper `MethodFilters` whose method `getPrivateMethods(Class)` calls `Class.getDeclaredMethods()` and filters out the non-private methods from the set returned by it. This seemingly harmless new functionality allows attackers to access *all private methods of all classes within the JCL*. The shortcut within `Class.getDeclaredMethods()` only considers the class loader of `MethodFilter`, which is the same loader that loaded all other system classes. The new wrapper method `getPrivateMethods()` thus exposes critical functionality to arbitrary callers with severe consequences.

In the best case, the developer of a caller method knows about the shortcut in the callee, e.g., through studying the Java Secure Coding Guidelines, which provides explanations and a list of methods that require special care. He may consciously decide to take the risk and the responsibility to prevent harm. When he does so, this decision is not documented in the code. In future code revisions, maintainers unaware of the special requirements imposed by the shortcut may inadvertently invalidate the security precautions taken by the original author.

Hard to analyze
Hardcoded shortcuts are hard to analyze, especially because there is no dedicated Java language construct or API support to express and document assumptions about the call stack. As a result, the effect and the scope of the implicit privilege elevation can only be reasoned about by careful examination of the shortcut's implementation and in addition requires deep knowledge of system classes and their properties. This reasoning is a very tedious and error-prone task. Thus, even when the developers know the list of methods that implement shortcuts by heart, using them implies constant awareness and a lot of effort by developers to prevent the introduction of new confused-deputy vulnerabilities.

Easy to break
It is hard to maintain the security of shortcut-containing code in the face of code evolution. Security-sensitive methods that implement shortcuts often assume a specific order of callers on the call stack. Changes to the code that affect the order of callers may cause the sensitive method to misbehave, if hardcoded assumptions are not properly adjusted. It is hard to judge whether a local change in the codebase violates the assumptions of some hardcoded shortcut implemented

in an entirely different location within the JCL. Thus, every change has to be properly analyzed to rule out potentially negative side effects on policy enforcement. Since, as already mentioned, such an analysis is manual and very involved, the risk is high that code evolution will introduce vulnerabilities.

We previously discussed the wrapper method `Util.openFileFromRoot()` in the example in Listing 5.2 on page 95 that illustrates how easy it would be to introduce a confused-deputy vulnerability through code evolution.

Inflexible
Hardcoded shortcuts are inflexible due to their very nature. Changes in the deployment environment for Java applications may affect risk considerations and security requirements. Adjusting policies accordingly is a matter of configuration, whereas changing hardcoded shortcuts is impractical.

Violating secure design principles
Shortcuts violate several well-accepted secure design principles. Yee [112] proposed a set of ten fundamental principles that should be followed when designing a secure system. While those principles were originally developed to reason about the usability of entire software systems from an end user's perspective, such as the user interface of a password prompt dialog, Türpe [107] showed that the same principles can also be applied for purposes of API usability evaluation. The deficiencies discussed above violate five of the ten principles:

- "Path of least resistance"—Developers need to invest extra effort to prevent the introduction of privilege escalation vulnerabilities.
- "Explicit authority"—Shortcuts implicitly grant privileges to a subset of callers, with them being potentially unaware of this fact.
- "Visibility"—Method signatures and other externally-visible properties of methods in system classes provide no indication of whether a method contains a shortcut.
- "Revocability"—Developers cannot refrain from privilege elevation through shortcuts, as there is no way to configure or express this.
- "Clarity"—The effects on policy enforcement are unclear when using a method that contains a shortcut.

5.3.3 Summary

To illustrate an extraordinary case of shortcuts, Listing 5.5 shows actual code that was released as part of `java.lang.SecurityManager` in Java 1.7.0_07. The code in this example was used by `java.lang.Class` to restrict reflective access from one class to members of another class. To this end, reflective methods like `Class.getDeclaredMethods()` and `Class.getDeclaredFields()` call a private method `Class.checkMemberAccess()`, which in turn calls `SecurityManager.checkMemberAccess()`. A shortcut in the latter method will bypass a call to `checkPermission` in line 20, thus preventing stack inspection and granting the privilege implicitly to selected callers of reflective methods in `java.lang.Class` if certain constraints

Listing 5.5: Real-world instance of a shortcut including inline comments in Java 1.7.0_07

```
1  public void checkMemberAccess(Class<?> clazz, int which) {
2    if (clazz == null) {
3      throw new NullPointerException("class can't be null");
4    }
5    if (which != Member.PUBLIC) {
6      Class stack[] = getClassContext();
7      /*
8      * stack depth of 4 should be the caller of one of the
9      * methods in java.lang.Class that invoke checkMember
10     * access. The stack should look like:
11     *
12     * someCaller                        [3]
13     * java.lang.Class.someReflectionAPI [2]
14     * java.lang.Class.checkMemberAccess [1]
15     * SecurityManager.checkMemberAccess [0]
16     *
17     */
18     if ((stack.length<4) ||
19       (stack[3].getClassLoader() != clazz.getClassLoader())) {
20       checkPermission(SecurityConstants.
             CHECK_MEMBER_ACCESS_PERMISSION);
21     }
22   }
23 }
```

on the call stack are satisfied. This is a conspicuous case because `SecurityManager.checkMemberAccess()` makes extensive assumptions about the call stack, involving the size of the stack and the order of callers, see lines 18 and 19. It may easily happen that code will be introduced that violates these assumptions, which is also why one finds several instances of the following warning in `java.lang.Class`: "be very careful not to change the stack depth of this checkMemberAccess call for security reasons".

Already in 2009, Li Gong [38] highlighted that counting stack frames is highly fragile and presented stack inspection as a key feature of Java 1.2 that would finally allow for more reliable access-control checks. The discussion above clearly shows that shortcuts add to the Java runtime's fragility by retaining many of the security issues that Java's current security model was supposed to resolve. We conclude that shortcuts significantly complicate the task of writing secure code in the first place and even more so the task of maintaining security in the face of code evolution. This claim is supported by various confused-deputy vulnerabilities in past versions of Java that we discussed in Chapter 3, which demonstrate how attackers can profit from inadvertently exposed functionality.

5.4 Proof-of-concept solution

The previous sections introduced the notion of shortcuts and explained how they contribute to various aspects of Java's insecurity. This section provides a proof-of-concept solution that is supposed to resolve these issues by moving from implicit privilege elevation through shortcuts to explicit privilege elevation through privileged blocks.

We implemented our proposed solution on the basis of OpenJDK 8[5] such that we can evaluate its feasibility and performance impact. All artifacts that are required to reproduce our results are publicly available.[6]

5.4.1 Overview

We propose a systematic tool-assisted hardening of the JCL that replaces instances of shortcuts by proper implementations of privileged blocks. This reduces the attack surface and virtually avoids the class of security-breaking programming mistakes that shortcuts enable.

For illustration, consider again the code examples in Listings 5.1 and 5.2. Both examples implement similar functionality with the difference of Listing 5.1 implementing a proper privileged block and Listing 5.2 implementing a shortcut instead. In both examples, we assume that `FileAccess` and `SystemProperties` were part of some release of the JCL, and that `Util` was introduced at a later point in time. The important difference between the two examples is that the introduction of the simple and exact same wrapper method `Util.openFileFromRoot()` leads to a severe security vulnerability in Listing 5.2 that uses shortcuts, while the code in Listing 5.1 remains intact due its proper use of privileged actions.

The goal of our systematic hardening is thus to eliminate shortcuts by replacing them with proper permission checks, and adjust callers accordingly so that they elevate privileges explicitly by a call to `doPrivileged()`. Adjusting callers this way is necessary to retain backward compatibility. Applying our hardening to `FileAccess` and `SystemProperties` in Listing 5.2 would result in code similar to Listing 5.1.

The following paragraphs provide more details and explain the various steps that are needed to implement our proposed solution.

5.4.2 Locating shortcuts

The first step in the transformation process is to find all methods in the JCL that implement a shortcut. The identification of these methods is complicated by three related factors. First, there is no dedicated language support to express constraints on the call stack, which is why they cannot be trivially recognized. Second, security-sensitive methods do not necessarily implement shortcuts and calls to the security manager themselves, but may use helper methods like `checkMemberAccess()` in Listing 5.5 instead. Third, security-sensitive methods are scattered all over the codebase, so the identification process has to take into account all parts of the

[5] OpenJDK 8 b132-03_mar_2014

[6] `https://github.com/stg-tud/jdeopt`

JCL. The JCL comprises a rather large codebase, which renders infeasible all purely manual approaches.

There is, however, one common property shared by all methods that implement shortcuts. They all make use of functionality to retrieve information about the current call stack, which is required to be able to check constraints on specific callers on the stack. The task of retrieving this information is typically not delegated to helper methods, as calling helper methods would yet again modify the call stack and thus add an additional layer of complexity. By manually reviewing shortcuts we already knew, we found that they either use `sun.reflect.Reflection.getCallerClass()` or `java.lang.SecurityManager.getClassContext()`. To our advantage, the number of methods in the JCL that provide information about the call stack is very limited.

We implemented a simple static analysis using the Soot framework [60] to find all methods that contain call sites for these two methods. We only used the Soot framework to located the specific bytecode instructions conveniently. The analysis does not need a call graph nor does it consider data flows. One could as well have used a text-based matching tool such as `grep`, but using Soot helps avoiding mistakes in the process.

Our static analysis yielded 86 candidate methods in total, which we reviewed manually to find the subset of methods that actually implement a shortcut. These are the results:

- Out of the 86 candidate methods, 35 do indeed implement a shortcut. They check constraints on the call stack and skip a permission check if these constraints are satisfied.
- Further six methods do not implement shortcuts in the strict sense, because they do not call a method of the form `SecurityManager.check*()` under any circumstances. Because of this, we consider them to be out of scope. They are noteworthy, however, because they deny access to functionality if the immediate caller's class loader is unable to load a specific class involved in the desired action. Such code implements a kind of undocumented poor man's approach to access control.
- One method does also not implement a shortcut in the strict sense, but it checks if the immediate caller's class loader is the bootstrap loader, and throws a `SecurityException` otherwise.
- The remaining 44 methods are caller sensitive, but review the stack for purposes other than shortcuts.

We matched our findings with the relevant sections in the Java Secure Coding Guideline, which provide a list of 75 caller-sensitive methods that have to be used with special care. The methods listed in the guideline constitute a subset of the 86 candidate methods we found by static analysis, which supports the appropriateness and accuracy of our approach. The additional eleven methods we found in addition to those that are also discussed in the guideline include nine methods that do not perform permission checks, one deprecated method, and one method which is part of `sun.misc.Unsafe` and hence not officially supported. From this, we conclude that the guideline is sufficiently complete with respect to shortcut-containing methods. For 41 out of the total 75 methods in the guideline we found no indication for shortcuts. In most cases, these methods implement dynamic access checks in the context of reflection, or provide dynamic loading capabilities involving the immediate caller's class loader. Both is caller-sensitive behavior that requires special attention from developers, and might even bring along a potential for vulnerabilities, which is also the reason why they are discussed in the guideline. We leave

these methods out of the scope of this chapter, since our focus is on shortcuts, but assume this may be worth further investigation in future work.

5.4.3 Removing shortcuts

Out of the 35 methods that we identified to implement a shortcut to bypass proper permission checks, we manually modified 32 of these methods to remove any conditionals that involved properties of the call stack, which may have prevented a permission check from being executed. We found that most shortcuts use `Reflection.getCallerClass()` to retrieve the immediate caller, and check if its defining class loader matches a specific type, or is `null`, i.e., the bootstrap class loader.

By removing shortcuts, we transformed 28 out of these 32 methods from caller-sensitive to caller-insensitive methods, guarding their functionality by a well-defined permission check. The remaining four methods remained caller sensitive after our modification, because—aside from their original shortcuts—they implement additional functionality, such as visibility checks in the context of reflective access. It is important to stress that caller sensitivity and the notion of shortcuts, as we defined it, are two separate concepts: Our notion of shortcuts always implies caller sensitivity, but the inverse does not always hold.

As stated above, we removed shortcuts from only 32 out of 35 methods that we found. One of the three remaining methods, `SecurityManager.checkMemberAccess()`, we decided to remove entirely from the codebase, because it is deprecated and not used by any other methods in the version of the JCL we used for our experiments.

The other two remaining methods, `Class.getDeclaredField()` and `Class.newInstance()`, could not be modified due to circular dependencies. After an initial attempt to modify them, we encountered errors during JVM initialization, because using either of the two methods causes a permission check, within which the method itself is called again, which in turn triggers another permission check, and so on. In the original code, the shortcuts in the two methods prevented this call sequence, because eventually `newInstance()` and `getDeclaredField()` would simply skip the permission check and succeed. We did not further investigate whether the use of reflection in the call sequence initiated by a permission check is inevitable. At the same time, we could not come up with a clean solution that would allow the shortcuts to be removed, without making substantial changes to the JCL. We thus decided to keep these two shortcuts and leave all callers of the two methods unmodified.

5.4.4 Adapting all callers

The last step of our proposed solution is to adapt all immediate calls to the methods we modified in the previous step. In the original code, many of the JCL's callers are able to access functionality only because of existing shortcuts that skip permission checks, even when there is untrusted code on the call stack. After modification, however, the same call sequences would fail because the permission checks that were previously skipped will now be executed, taking into account the full call stack. To retain backward compatibility to the largest possible extent, we thus have to wrap all immediate calls to previously shortcut-containing methods in privileged actions. As we

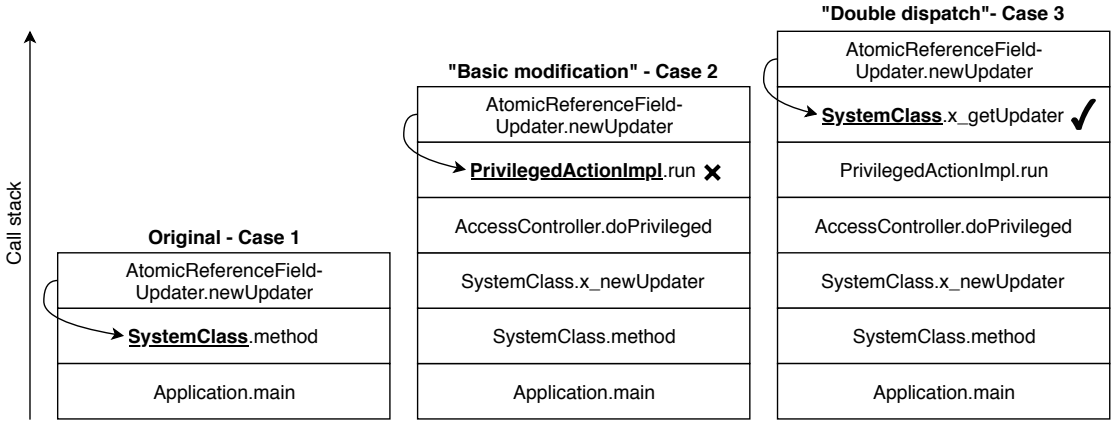


Figure 5.2: Different modification strategies and their effect on the call stack

will explain, the calls to modified methods have to be adapted differently, depending on whether the modified method is still caller-sensitive after modification.

Modifying calls to the 28 methods that lost their caller sensitivity through modification works as follows. First, we use static analysis to find all immediate calls to any of the modified methods. For this, we adapted and reapplied the approach we used to locate shortcuts as described above. As a result, we found 1,399 calls in the JCL that required modification. For one out of the 28 modified methods, we were not able to find even a single caller within the JCL itself, which means that our transformation regarding this method is already complete at this point.

We then used Javassist [16] to implement a bytecode modification tool that automatically adapts all calls. It adds one or more private helper methods to each calling class, each of which instantiates a privileged action that wraps the original target method call and then calls `AccessController.doPrivileged()`. Next, the modification redirects all calls targeting a modified method to one of the newly added helper methods. Note that each helper method wraps a call to one specific modified method only, which is why multiple helper methods are added to calling classes that target more than one modified method. Privileged actions have to be implemented in separate classes, for which application developers often use anonymous inner classes, as shown in Listing 5.1. However, instead of adding one or more individual implementations of privileged actions for each calling class, we added a small set of commonly accessible privileged actions to `java.lang.Class`, shared among all helper methods. By this, we avoid having to add hundreds of additional classes to the JCL which would unnecessarily bloat the codebase.

We decided for bytecode modification instead of source-code modification because at the time we conducted our experiments, we were not aware of any publicly available source-code modification libraries that would have allowed us to perform the required modifications in an automated fashion.

The calls to one of the four methods that remained caller sensitive after the removal of shortcuts had to be modified differently. This is because those methods vary their behavior depending on the immediate caller, which, as can be seen in "Case 2" in Figure 5.2, would be the `run()` method of a privileged action if we were to modify the callers in a similar way as described

above. Our basic approach would thus result in undesired behavior, as the caller-sensitive target method would vary its behavior depending on the wrong caller.

Out of the four methods that remained caller sensitive, only one method, `AtomicReferenceFieldUpdater.newUpdater()`, is actually called from other methods in the JCL. Due to the fact that also this method is called from only three other methods in the entire class library, we decided to modify these callers manually. To this end, we used a modification approach that implements a form of double dispatch, see "Case 3" in Figure 5.2. First, we manually added two private helper methods, `x_newUpdater()` and `x_getUpdater()`, to each calling class of `AtomicReferenceFieldUpdater.newUpdater()`. The method `x_newUpdater()` instantiates a privileged action whose `run()` method calls `x_getUpdater()`. In turn, `x_getUpdater()` calls `AtomicReferenceFieldUpdater.newUpdater()`. Finally, we replaced all original calls to `newUpdater()` by calls to `x_newUpdater()`. The effects of this alternative modification strategy on the call stack can be seen in "Case 3" in Figure 5.2. By routing the call sequence through `x_getUpdater()`, instead of immediately calling `newUpdater()` in the privileged action, we ensure that the immediate caller of `newUpdater()` is the original calling class, not the privileged action. As `newUpdater()` varies its behavior depending on its immediate caller's *class* and not the specific calling *method* it behaves appropriately, i.e., as before our modifications.

5.4.5 Benefits

Removing shortcuts from Java's codebase and adjusting callers accordingly as describe above strengthens the Java runtime's security architecture. The following explains that the benefits of our proposed solution are twofold. On the one hand, the resulting JCL code is easier to maintain, and in consequence it will be harder to introduce new confused-deputy vulnerabilities in future versions of Java. On the other hand, some of the existing attack vectors that depend on shortcuts will become infeasible.

Enabling security-preserving code evolution

The benefits with respect to facilitating security-preserving evolution of the JCL were already highlighted in Section 5.4.1 by discussing the code in Listing 5.2 and the result of the adaptation by our approach in Listing 5.1. The desired positive effect of our conversion is that now, if an unprivileged attacker calls `openFileFromRoot()`, the permission check will fail, because `Util`, having been added in a later release of the JCL, was not subject to our modification. This prevents the previous vulnerability.

By trading implicit elevation of privileges with shortcuts for explicit privilege elevation with privileged blocks as described above, we retain backward compatibility to a large extent. This has, however, a downside: It will retain confused-deputy vulnerabilities that already existed in the JCL at the time of its modification. Consider again the example code in Listing 5.2. If the vulnerability caused by `Util` is already part of the codebase at the time we apply our program transformation, the call to `openFile()` in `openFileFromRoot()` in line 23 will be wrapped in a privileged action, just like the call in `readProp()` in line 15. As a result, the `openFileFromRoot()` method will continue to expose critical functionality to attackers even after program transformation.

It is not easy to decide which method calls under privileged regime are legitimate, and which ones represent a vulnerability. With the current proof-of-concept implementation of our approach, we reduce the possibilities of potentially illegitimate privilege elevation to explicit ones only, but still leave the identification of insecure uses of critical functions out of the scope.

Rendering existing attack vectors infeasible
An important beneficial aspect of our proposed solution is that, even in its current state of development, our transformation renders existing attack vectors infeasible, thus effectively blocking certain attacks. This is, because many attacks that exploited previously shortcut-containing methods did not call these methods directly, like `Util`, but rather by abusing an insecure use of reflection or method handles. The proposed transformation does not modify such kinds of calls, as privileged blocks are only introduced for direct callers. Therefore, after the shortcuts are removed, any such attack will fail: The permission check in the reflectively called method, from which the shortcut was removed, will now trigger a proper permission check, preventing the action if the call sequence was initiated by untrusted callers.

As already mentioned, we found four examples[7] of such kinds of attacks in a sample set of exploits provided by Security Explorations, see Chapter 3. They leverage vulnerabilities involving the insecure use of reflection to call shortcut-containing methods through a trusted system class. We verified through debugging that performing permission checks instead of taking the shortcuts will result in access-control exceptions, thus effectively preventing these attacks.

Interestingly, after Security Explorations reported three of those four vulnerabilities[8] to the vendor, a fix was released that did not adequately address the underlying security issues [98]. One of the problems was that the fix release still allowed an attacker to use the original vulnerabilities to call shortcut-containing methods and other caller-sensitive methods. As a consequence, Security Explorations was able to adapt three of the four original exploits by minor edits so that they would work again on the fix release.

In conclusion, these findings demonstrate that our proposed solution does increase the security of the Java platform, and also support our claim that shortcut-containing code is hard to maintain.

5.5 Performance evaluation

We argued that removing shortcuts from the JCL as we proposed increases the Java runtime's resistance against attacks and also increases the maintainability of its codebase. However, one of the main reasons for why shortcuts were introduced in the first place was to reduce the runtime cost of access control checks by skipping permission checks [19].

In this evaluation we hence address the following research question: *Which runtime overhead does the code adaptation introduce?*

[7] Issues 32, 35, 36, and 37
[8] Issues 35, 36, and 37

5.5.1 Evaluation setup

We implemented our proposed solution on the basis of OpenJDK 8[9] and used this modified Java runtime for several experiments. As a baseline we used the same version of the OpenJDK without modifications. To ensure maximum comparability, we built both the modified and the unmodified version ourselves based on the official source release.[10]

We compared both variants in two different settings. In the first setting, we conducted a set of macro benchmark tests. We used the DaCapo benchmark suite [12] version 9.12-bach to perform tests on both variants of the Java runtime. The goal is to measure the relative overhead that the transformations may induce in the execution of real-world applications. We chose DaCapo because it consists of complex, real-world applications from diverse application domains that cover a broad range of possible program behaviors [12]. Using DaCapo's built-in functionality, we implemented a custom callback class that performs 250 timed runs for each benchmark in each setting, preceded by 750 warm-up runs. We chose such a high number of iterations to minimize the effects of outliers that can be caused by just-in-time compilation or other reasons. By this, we maximize reproducibility of our results and ensure that comparing runtime values is actually meaningful. The following command was used to execute the tests:

```
java -Xcomp -XX:CompileThreashold=1 -server -Xmx2g -Xms2g -Xbatch -cp ".;./
    mathlib.jar;./dacapo.jar" Harness -t 1 -c callback {benchmarkname}
```

Due to a known bug[11] in the version of DaCapo we used, we had to measure `jython` runtimes without `Xcomp`-flag. We were further required to entirely skip the two benchmarks `batik` and `eclipse` because their execution resulted in errors on both the original and modified OpenJDK. Specifically, `eclipse` failed during its checksum validation, indicating that the benchmark produced an unexpected output. We were able to reproduce this problem with multiple original Java runtimes on different machines and have reported this problem to the benchmark's authors. The execution of the second failing benchmark results in a `ClassNotFoundException`, apparently because it accesses a class that is available in Oracle's JRE, but not in the OpenJDK.

In addition to runtime measurements, we also counted the number of method calls to any of the 32 modified methods that are performed while executing the DaCapo benchmarks. This allows us to reason about the coverage of the DaCapo suite with respect to the proposed changes.

In the second setting, we ran both variants of the OpenJDK on micro benchmarks. These micro benchmarks are artificial test scenarios that we created for all transformed code locations. The goal is to assess the transformed code locations without measuring influences by unaffected code. We do this by calling the first publicly accessible method that transitively calls a modified method. Evaluating our code adaption using such micro benchmarks provides insight into how much the removal of shortcuts costs in the worst case, by calling modified methods frequently. To measure runtimes we used JUnitBenchmarks 0.72.[12] For each modified method, we have created a dedicated JUnit test case which contains minimal setup code, code to prevent dead code elimination [36], and a loop that performs ten million calls to the method whose runtime is to be

[9] OpenJDK 8 b132-03_mar_2014

[10] `https://download.java.net/openjdk/jdk8/`; last accessed 19-Dec-2018

[11] `https://sourceforge.net/p/dacapobench/bugs/80/`; last accessed 19-Dec-2018

[12] This project has been discontinued after we conducted our study

Table 5.1: Runtimes of DaCapo in seconds

Benchmark	Security manager disabled				Security manager enabled			
	Original	Modified	Overhead abs.	Overhead rel.	Original	Modified	Overhead abs.	Overhead rel.
avrora	3.08 ±[0.15]	3.11 ±[0.10]	0.03	1%	3.02 ±[0.06]	3.06 ±[0.02]	0.04	1%
fop	0.31 ±[0.01]	0.31 ±[0.01]	0.00	1%	0.32 ±[0.01]	0.32 ±[0.01]	0.00	1%
h2	3.70 ±[0.01]	3.70 ±[0.01]	0.00	0%	3.67 ±[0.01]	3.70 ±[0.01]	0.00	1%
jython	1.48 ±[0.02]	1.47 ±[0.02]	−0.01	−1%	1.50 ±[0.02]	1.47 ±[0.03]	−0.03	−2%
luindex	1.02 ±[0.12]	0.99 ±[0.05]	−0.03	−2%	1.20 ±[0.07]	1.21 ±[0.08]	0.01	1%
lusearch	4.95 ±[0.02]	4.95 ±[0.02]	0.00	0%	4.99 ±[0.01]	4.86 ±[0.02]	−0.13	−3%
pmd	2.50 ±[0.02]	2.48 ±[0.02]	−0.02	−1%	3.06 ±[0.03]	3.05 ±[0.04]	−0.01	0%
sunflow	8.36 ±[0.03]	8.31 ±[0.02]	−0.05	−1%	8.36 ±[0.04]	8.34 ±[0.03]	−0.02	0%
tomcat	48.54 ±[0.28]	48.56 ±[0.31]	0.02	0%	52.54 ±[0.81]	52.35 ±[0.35]	−0.19	0%
tradebeans	8.71 ±[0.02]	8.83 ±[0.03]	0.12	1%	10.01 ±[0.05]	10.02 ±[0.02]	0.01	0%
tradesoap	17.94 ±[1.27]	17.67 ±[0.89]	−0.27	−2%	23.22 ±[1.44]	23.54 ±[2.02]	0.32	1%
xalan	6.54 ±[0.04]	6.60 ±[0.04]	0.06	1%	6.76 ±[0.03]	6.79 ±[0.02]	0.03	1%
				∅ −0.25%				*∅ 0.08%*

measured. This high number of iterations is required because a single invocation is too fast to be measured accurately. JUnitBenchmarks computes the average and standard deviation of the runtime of ten rounds, i.e., $10 \times 10,000,000$ calls to the target method, and the ten rounds are preceded by five warm-up rounds, i.e., $5 \times 10,000,000$ calls to the target method, not included in measurements. The standard deviation is used to gauge the accuracy of the results.

Lastly, we perform both experiments using two different setups. In the first setup, we perform the experiments without enabling a security manager. This setup acts as our baseline. In the second setup, we execute the experiments with a security manager set programmatically, in case of micro benchmarks, and by command line argument, in case of DaCapo, using VM arguments `-Djava.security.manager -Djava.security.policy`. We use the security manager with a policy file granting all permissions to all code. We manually verified that enabling the security manager this way actually triggers permission checks at runtime, despite the fact that the code has effectively the same permissions as if no security manger were present.

All experiments were performed on a machine with an Intel Core i5-2400, 3.1 Ghz processor, with 4 GB of memory, running 64-bit Windows 7 Enterprise SP 1.

5.5.2 Results of macro benchmark tests

Table 5.1 shows the results of executing the DaCapo benchmarks suite on the original and modified version of the OpenJDK. Each benchmark of the suite is represented by one row in the table. Column 1 shows the benchmark's name. Columns 2 and 3 show the execution time in seconds without security manager in place, along with the standard deviation for each value. Column 4 shows the runtime difference as a factor to highlight the cost of our code adaptation. Columns 5, 6, and 7 show the same values measured with the security manager enabled.

Comparing the runtimes of the original code and the modified code, in almost all cases the relative difference lies below 1%. In three cases, measurement results suggest that the modified

Table 5.2: Call statistics for DaCapo

Benchmark	# Calls	# Methods	Most frequently used
avrora	147	11	`getClassLoader()`
fop	6,329	11	`getContextClassLoader()`
h2	210	11	`getClassLoader()`
jython	1,483	19	`getMethod()`
luindex	208	11	`getClassLoader()`
lusearch	159	11	`getClassLoader()`
pmd	1,885	12	`getClassLoader()`
sunflow	249	12	`getClassLoader()`
tomcat	32,069	17	`getDeclaredMethods()`
tradebeans	219,792	22	`getContextClassLoader()`
tradesoap	668,000	22	`getFields()`
xalan	36,396	11	`getParent()`
	∅ *80,602*	∅ *14*	
	Σ *967,227*	∪ *24*	

code is faster by 2%. In one exceptional case, the adapted code appears to be 3% faster. Taking all results into account, the modified code is at most 1% slower than the original code. We attribute these small runtime differences mostly to instabilities of the JVM [36] rather than to the changes we applied to the code, and can confirm the observation of Gil et al. that even testing identical code may lead to slightly different results in terms of runtime. The execution speed is further influenced by secondary factors induced by the underlying software and hardware stack, as previously studied by Gu et al. [46].

In addition to runtime measurements, we also collected call statistics to ensure that the methods we modified during our adaptation are actually involved in benchmark execution. Our results clearly show that this is the case. Table 5.2 shows a summary of the results we measured for the original OpenJDK without a security manager enabled.[13] Each of the twelve DaCapo benchmarks we used in the experiments is represented in one row. Column 1 shows the benchmark's name, column 2 shows the total number of method calls to any of the modified methods, column 3 shows the number of modified methods the benchmark uses. Finally, column 4 shows the modified method that was most frequently used by the respective benchmark

As can be seen from the results, the DaCapo benchmark suite extensively uses most of the 32 methods under investigation. Running any of the benchmarks requires at least eleven out of 32 methods, and 22 at most. Only eight out of 32 modified methods are not used at all by DaCapo. Further, executing just a single run of one of the benchmarks involves between 147 and 668,000 calls to modified methods. A single run of the entire benchmark suite requires more than 900,000 calls to the modified methods.

[13] Complete call statistics are provided with the artifacts

Table 5.3: Summary of runtimes of micro benchmarks in microseconds (µs) per single invocation

	Security manager disabled			Security manager enabled		
	Original	**Modified**	**Overhead**	**Original**	**Modified**	**Overhead**
Min.	<0.01	<0.01	-0.27	0.03	0.58	0.22
Max.	31.70	49.25	17.55	79.15	99.65	20.50
Avrg.	1.94	2.94	1.00	3.80	6.18	2.38
Med.	0.29	0.16	0.00	0.45	2.05	1.17

Summarizing the results, we conclude that the proposed code changes have virtually no performance impact on the tested real-world applications, even though the modified methods are heavily used.

5.5.3 Results of micro benchmark tests

As explained in Section 5.4.3, we removed shortcuts from 32 methods in the JCL. For all these methods, we implemented a micro benchmarks using JUnitBenchmarks. These micro benchmarks perform ten million calls each to the methods they target. In two exceptional cases, however, we reduced the number of calls to 200,000 due to the long runtimes of the target methods, and interpolated the results to allow for easy comparison.

Table 5.3 shows a summary of the results of the micro benchmarks. To avoid misunderstandings in the following discussion, all results are shown in microseconds (µs) per single invocation, instead of seconds per ten million calls. Column 1 shows the minimum, maximum, average, and median runtimes of the original OpenJDK without a security manager enabled. Column 2 shows the respective results for the adapted OpenJDK. Column 3 shows minimum, maximum, average, and median values of the absolute runtime differences between the original and adapted code. Columns 4, 5, and 6 show the respective results for the tests we performed with the security manager enabled.

Our first observation is that all methods under investigation execute extremely fast, and that is before and after adaptation. As can be seen in columns 1 and 2, without a security manager enabled, a single call to the fastest method completes in <0.01 µs, both on the original and modified OpenJDK. The slowest method requires 31.7 µs on the original code, and 49.25 µs on the adapted code, which equals an overhead of 17.55 µs.

For the tests performed with the security manager disabled, this is the highest absolute impact that we encountered. The result set contains one more outlier with an overhead of 12.7 µs, while the remaining 30 methods show differences in runtime between −0.27 µs and 1.82 µs. In fact, in this setting only 13 out of 32 methods show a performance penalty at all. Based on the call statistics we collected for DaCapo, we can say that one of the two outliers is not used at all, while the other one is used more than 16,000 times in the entire suite.

On average, with the security manager disabled, the original OpenJDK executes the methods under investigation in 1.94 µs per single call. The modified OpenJDK takes 2.94 µs, which is an average overhead of 1 µs per method call induced by our code adaptation. It is important

to note, however, that the two outlier methods mentioned above influence average values to a greater extent that all the other methods. The median execution time for a single method call without security manager on the original OpenJDK is 0.29 µs, compared to 0.16 µs on the adapted OpenJDK, demonstrating that several methods even became faster. Overall, the runtimes we measured without the security manager enabled show that, if at all, there are only insignificant performance penalties induced by our proposed changes.

The performance measurement results for tests performed with the security manager enabled are not much different to the results we collected with the security manager disabled. As can be seen in columns 4 and 5 in Table 5.3, the minimum execution time increased from 0.03 µs per single method call to 0.58 µs. The method that originally completed in 0.03 µs shows the largest relative overhead induced by our code adaptation. It needs 2 µs after modification and is thus not the fastest method in the adapted OpenJDK anymore. It is one of four outliers in this setting with a relative overhead >1000%: `getParent()`, `getContextClassLoader()`, `getClassLoader()`, and `getSystemClassLoader()`. In all these cases, however, the original execution time was significantly <0.1 µs per single method call, and <2.1 µs after adaptation, which we still consider extremely fast. As can be seen in Table 5.2, three out of those four outliers are the most frequently used modified method by at least one of the benchmarks. We can thus say that DaCapo provides good coverage in that respect.

In terms of absolute overhead, the result set includes two outliers with an overhead >10 µs per single call, while the remaining 30 methods show overheads between 0.2 µs and 4 µs. Those two outliers are the same methods that showed the largest absolute overheads observed with the security manager disabled. One of those two outliers is the longest running method before and after adaptation. As can be seen in the second row in columns 4 and 5 in Table 5.3, it has a runtime of 79.15 µs per single call in the original OpenJDK, and 99.65 µs in the adapted OpenJDK. These two outliers, as discussed before, substantially influence the average value of the collected runtimes. With the security manager enabled, the average runtime increases from 3.8 µs per single method call to 6.18 µs, which is an average overhead of 2.38 µs. The median runtime increases from 0.45 µs to 2.05 µs per single method call, and the median overhead is 1.17 µs. None of the methods under investigation became faster through our adaptation in a setting that has the security manager enabled.

In summary, we can observe measurable performance penalties when assessing the runtimes of modified methods in isolation. However, these penalties are very small in absolute terms, which is why they do not influence the runtimes of the real-world applications contained in the DaCapo benchmark suite.

5.5.4 Discussion

We explained in Section 2.2.1 that Java checks permissions by evaluating whether all callers in the concerning access control context belong to protection domains that were assigned sufficient privileges. Unless explicitly implemented differently using custom instances of `AccessControlContext`, the access control context contains the set of callers that are currently on the call stack when the permission check is triggered. Privileged blocks can be used to reduce

the set of callers that belong to the access control context to allow that certain well-defined, security-sensitive actions can be triggered even by untrusted code, see Figure 2.6.

Our proposed solution removes shortcuts that check properties of only the immediate caller, which causes the execution of thorough permission checks. We thus introduced privileged blocks to compensate for this change, by which we retain backward compatibility. Due to the fact that we wrap all *immediate calls* to previously shortcut-containing methods in the JCL in privileged actions, we effectively reduce the set of callers within the respective access control contexts to an absolute minimum—the exact number depends on whether we applied the "basic modification" or the "double dispatch" approach to modify the respective method, see Figure 5.2. In any case, all callers in this reduced access control context belong to the same protection domain, the system domain, because our modifications only apply to classes within the JCL.

Checking permissions for a single protection domain is trivial and it is thus reasonable to assume that, with current technology, even a shortcut implementation is not considerably faster than a proper permission check when executing real-world applications on our adapted Java runtime. The macro benchmark tests we performed with the DaCapo benchmark suite support this claim.

5.6 Productive use and further research

Removing shortcuts from Java's codebase as we propose hardens the runtime's implementation of access control. The extensive performance evaluation we conducted shows that the changes we propose have no negative impact the execution speed of a set of complex real-world applications. However, implementing our proposed solution for productive use requires reconsideration of two important aspects, as the following explains.

5.6.1 Adjusting security policies

The code adaptation we performed to remove shortcuts from methods in the JCL retains the runtime's compatibility with legacy applications. This means that all applications that were implemented and compiled for the original runtime can also be executed on the modified runtime. However, the policies of some legacy applications may have to be adjusted when switching from a shortcut-containing runtime to a shortcut-free runtime.

An application's privileges are usually defined by a set of permissions granted explicitly in a security policy. This is not the case for privileges gained through shortcuts, because they are hardwired into the JCL. Removing shortcuts will cause permission checks to be executed that would have been skipped otherwise, which will require permissions that were not needed before the adaptation. Some legacy applications will thus require adjustments to their security policy when upgrading to runtime environment that is shortcut-free. This task can either be done manually by determining the required permissions through code reviews and dynamic testing, or automatically by means of static analysis that computes the set of required permissions for any given application class. Appropriate approaches have been proposed earlier [59].

This issue does not concern all legacy applications. No changes to the security policy are required for applications that do not immediately call shortcut-containing methods, call them in a

way that does not trigger a shortcut, have already been granted all required permissions anyway, or run without a security manager.

5.6.2 Reworking Java's standard permissions

Java's standard set of permissions that are part of the JCL cannot equivalently express all privileges gained through shortcuts in terms of their semantics. As an example, a shortcut in `Class.getDeclaredFields()` skips a permission check if callers attempt to access fields of classes that were loaded with the same class loader. Removing the shortcut in `getDeclaredFields()` will cause it to trigger a permission check that checks for the *RuntimePermission 'accessDeclaredMembers'*. As explained above, the security policies of legacy applications can be updated to grant this permission when upgrading to a shortcut-free runtime environment with little to no effort. However, the problem is that granting this permission provides applications with *more* privileges than the original shortcut implementation—it allows callers to access fields of *arbitrary* classes, including private members of system classes.

If, for instance, an application class uses `getDeclaredFields()` to access its own fields, or the fields of classes that were loaded by the same application class loader, no explicitly granted permission is needed in the presence of shortcuts. The application class would be prevented from accessing fields of, e.g., system classes, because they were loaded by another class loader. Removing the shortcut from `getDeclaredFields()` would prevent the application class from accessing any fields unless being granted *RuntimePermission 'accessDeclaredMembers'*. Granting this permission, however, would allow the application to access members of all classes.

Note that the security implications of this are limited because *RuntimePermission 'accessDeclaredMembers'* only allows for retrieving instances of `java.lang.reflect.Field` and `java.lang.reflect.Method`, but using these instances to read and write field values, or call methods requires yet another permission.

This is nevertheless an issue, and it is caused by the fact that some standard permissions are too coarse-grained to be used in a meaningful manner. Permissions that are supposed to restrict the use of reflection only allow for on/off decisions, thus either allowing reflective access to all available classes, or none at all. The consequence of coarse-grained standard permissions is that, when upgrading to a shortcut-free runtime, applications may have to be granted permissions that provide more privileges than originally granted through shortcuts. This is inherently risky, as it violates the principle of least privilege. Even outside the context of our proposed changes we consider permissions like the reflection permissions as too coarse to be really useful in a security-sensitive setting.

This circumstance is not caused by technical issues, but is simply a design flaw. Consider, for comparison, Java's standard file permission, which, in contrast to the reflection permissions, provides great flexibility for fine-grained access decisions. It allows for specifying the file path to which the permission applies, as well as the specific actions that shall be granted, such as 'read' or 'write'. Similar expressiveness is desired for securely restricting the use of reflection and method handles by untrusted applications, and for adequately compensating for the removal of shortcuts.

We thus argue that a thorough redesign of Java's standard permissions is both possible and required. This is a complex task in itself that needs to take into account technical aspects, as well as various organizational and human factors. One of the major challenges is to allow for fine-grained access-control decisions that support the principle of least privilege, without being hard to use of performance-wasting. Further, the permission model should be designed to better support automatic policy generation for existing applications. We hope that future research will take on the challenge of developing a permission model that is both flexible, and usable within the settings that it is designed for.

5.7 Lessons learned

The rigorous analysis of shortcuts in Java's implementation of access control sheds light on how the Java runtime's security architecture is weakened in practice. Besides the impact our results may have on further developments of the runtime, we can also view this work as a case study and derive a set of general recommendations for the development of secure software. In the following, we will highlight general lessons learned, that hopefully serve as a guidance for the design and implementation of other complex security models.

Use explicit and clear privilege elevation
Our research clearly shows that by elevating privileges explicitly through constructs such as privileged blocks, one can avoid the accidental re-exposure of those privileges to attackers. One reason is that privileged blocks elevate privileges temporarily and only within a given lexical scope. Any code refactorings performed will move the privileged blocks along with the other code, causing privileges to be raised only where required. A second reason, though, is the pure presence of privileged blocks. To JCL maintainers it not only serves as a security construct, but also as a warning flag: Privileges are elevated at this point and need to be properly protected from being leaked to the outside.

Stick to the security model
Security models of complex systems are planned and designed prior implementation. Inconsistencies between design and implementation can be risky as they hamper proper evaluation and maintenance. In the concrete case we studied here, shortcuts are used instead of proper stack-based access control, which is a deviation from the Java security model that increases the attack surface. A common practice in software engineering is to readjust a project if it drifts apart from reality. It should be just as normal to readjust and re-evaluate a security model if strictly implementing it as prescribed is not possible, e.g., due to performance constraints. In the specific case of Java, had our evaluation been performed earlier, one would probably have had the chance to design a more fine-grained policy system in the first place, which then in turn would have allowed all current use cases without having to opt for implicit privilege elevation.

Thoroughly document tradeoffs between security and performance
Design and implementation of software is shaped by functional and non-functional requirements.

Tradeoffs are often necessary due to conflicting requirements, and security-related functionality not always has the highest priority. While in many cases at least functional requirements are documented, it seems less common to properly document how tradeoffs shaped the design and implementation of a complex software. In the specific case of shortcuts in Java's access control mechanisms, we were required to perform manual reviews, functional tests, and also double-check with representatives from Oracle to verify our assumption that one reason for which shortcuts were introduced was for performance reasons. In result, performance-related tradeoffs in long-living systems should be thoroughly documented, as performance constraints definitely change over time and many optimizations become obsolete eventually.

Re-evaluate performance tradeoffs in regular intervals
The very nature of a tradeoff is to balance out a negative impact with a positive impact of similar or higher value. If the hoped-for positive impact is improved performance, and the negative impacts are, e.g., an increased attack surface and decreased maintainability, then the value of this tradeoff changes over time as performance gains eventually decrease with optimizations of the runtime and hardware. We thus argue that a re-evaluation of performance-related tradeoffs in regular intervals should be part of the maintenance process of any long-living system.

5.8 Related work

Aside from Li Gong's extensive work [39, 38, 37] on the Java security model, several researchers took up the challenge of analyzing, extending, or breaking it. The following presents prior work that is concerned with analyzing various aspects of stack inspection, as well as work that strives for alternative approaches to design and implement access control in Java.

Analyzing stack inspection
In our study, we remove shortcuts and introduce privileged blocks instead to retain backward compatibility. However, as we explain in Section 5.4.5, by this we also retain confused-deputy vulnerabilities that were already contained in the codebase at the time we perform our transformation. Pistoia et al. [90] presented an interprocedural bytecode analysis approach that provides guidance on where to place privileged blocks, and also detects unnecessary or insecure implementations of privileged blocks. This approach can potentially be used to identify insecure privileged blocks in the transformation process that we propose to remove shortcuts from the JCL in order to further reduce the number of confused-deputy vulnerabilities in the Java runtime.

Koved et al. [59] presented a data flow analysis approach that can be used to determine the access rights required by a given Java application or library. We explained in Section 5.6.1 that this approach can potentially be used to automatically adjust the security policies of legacy applications when upgrading to a shortcut-free runtime.

Herzog et al. [49] analyzed the performance overheads of permission checks in the Java runtime. They provide practical guidance on how to use the access control mechanism efficiently, but they do not explicitly address the topic of shortcuts.

Cifuentes et al. [19] discussed in-depth how unguarded caller-sensitive method calls introduce security vulnerabilities, and provide practical guidance on how to avoid and detect instances of this security defect in the context of the Java security model.

Fournet and Gordon [34] provide formal semantics and an equitional theory for the general concept of stack inspection. Their work is not specific to any runtime implementation, such as Java or .NET, and does not address shortcuts.

Alternative approaches and designs to Java's access control mechanisms
In our study, we propose to remove shortcuts that skip permission checks, which potentially increases the number of checks that are performed during program execution. Several approaches have been developed to reduce the cost of stack inspection in the Java runtime, which are potentially suitable to complement our proposed solution. Bartoletti et al. [10] presented two high-performance control-flow analyses that approximate the set of permissions granted or denied to callers, which can be used to speed up permission checks. Chang [14] presented a technique to approximate permission sets that aims for a higher precision than previous approaches.

Other work puts a focus on increasing the maintainability of Java's implementation of access control. Toledo et al. [106] observed that access-control checks are scattered across the entire JCL, which, as they argue, decreases its maintainability. They propose a solution to modularize access control by using aspect-oriented programming.

Several alternative approaches to stack-based access control were presented in the past. Abadi and Fournet [5] presented history-based access control. In contrast to Java's approach of stack inspection, their approach not only considers the methods that are currently on the call stack, but also all methods that already completed execution in the past. Pistoia et al. [89] presented a more precise approach that considers only the subset of methods that are actually involved in a security-sensitive operation.

Wallach et al. [108] formally modeled the semantics of stack inspection using belief logic and presented a semantically equivalent approach to implement access control in Java. They argue that their approach is slower than Java's original implementation, but theoretically sound and easily adaptable to other programming languages.

5.9 Conclusion

Several security-sensitive methods in the JCL implement shortcuts, which use hardcoded heuristics to determine whether a proper permission check can be omitted. By this, shortcuts implicitly elevate the privileges of certain callers. However, due to this implicit nature of shortcuts, callers are often unaware of privilege elevation, and developers are thus likely to unknowingly introduce confused deputy vulnerabilities when extending or maintaining the codebase.

We thoroughly studied this concept in depth and showed that shortcuts significantly weaken the Java security architecture in practice. Specifically, they directly enable attack vectors and substantially complicate the task of security-preserving code maintenance.

Through a tool-assisted adaptation we have created a new variant of the JCL that works almost without shortcuts, allowing privileges to be elevated only explicitly through the use of

privileged blocks. The adapted code allows maintainers, security experts and tools to easily identify points of privilege escalation. Moreover, some previous exploits that abuse insecure use of reflection are effectively mitigated by our approach.

The primary reason for which shortcuts were originally introduced was to lower the execution overhead of access control. A large-scale set of experiments with the DaCapo benchmark suite shows that our proposed solution induces virtually no runtime overhead when executing a set of complex real-world applications on current hardware. Micro benchmarks explain this result by showing that even in the worst case the absolute overheads are all in the order of microseconds.

Besides an extensive performance evaluation, we further assessed in detail the usability implications of upgrading from implicit to purely explicit privilege elevation. The tradeoffs we discussed will ultimately determine whether the proposed hardening is worthwhile adopting at this point in time. We reported our findings to Oracle and engage in a continuous exchange to investigate this matter further. Oracle supports this ongoing work with a dedicated research grant.

6

Conclusion

In this final chapter, we conclude our research on the Java security architecture. To this end, we first summarize the most important aspects of this work and revisit the thesis statements already presented at the beginning of this work. Then, we put our findings in context with other systems and technologies to discuss the applicability of our results beyond the original scope, and further discuss general questions concerning the secure design of complex systems. Finally, we present directions for future work.

6.1 Summary

The Java programming language and runtime environment had a strong influence on software engineering as a practical discipline. Today, Java is one of the most popular development languages and it is applied in a broad range of different application contexts, including server-side applications, client implementations, mobile and embedded devices. The JRE is one of the first widely-used runtime environments that was designed from the ground up to provide strong security guarantees when executing untrusted code obtained from external sources. To this end, it implements a complex security architecture that enforces security policies in such a way that untrusted code can run along trusted code within the same process, using higher-level concepts such as stack-based access control and automatic memory management to restrict the rights of untrusted code. However, over the course of its entire lifespan, a large number of attacks revealed many severe security vulnerabilities in the JRE that allowed for a full bypass of all security mechanisms.

Despite the many examples of security vulnerabilities in the platform, only little was previously known about conceptual commonalities of different exploits and the extent to which design weaknesses in the Java security architecture enabled the attacks. Thus, in this work, we systematically collected and analyzed a large body of exploits for different versions of the JRE, covering vulnerabilities of more then ten years. To facilitate systematic analysis, we developed a new meta model that we used to define reusable building blocks, which in turn we used to document the behavior of all exploits in the sample set. This structured documentation enabled us to identify conceptual commonalities between the different exploits, and also to identify design weaknesses in the JRE's architecture. One result of this analysis is that there is a set of nine commonly abused weaknesses, including, e.g., the unauthorized use of restricted classes, the use of confused deputies, and abuse of caller-sensitive methods. Further, we showed that all exploits in

the sample set can be divided into three categories of attacks: single-step attacks, restricted-class attacks, and multi-step attacks. Finally, we identified two major design weaknesses that enabled many of the attacks. The first weakness is improper access control, which is manifested in various ways. On the one hand, this concerns caller-sensitive methods in publicly accessible system classes that skip proper, stack-based permission checks if the immediate caller is a trusted system class. On the other hand, this concerns the large number of restricted classes that entirely refrain from stack-based access control, thus relying on the platform to prevent access to these special classes through other means. The second design weakness besides improper access control that we identified was weak information hiding. Specifically, we found that the security of the entire Java platform rests on the confidentiality and integrity of specific variables in system classes, while at the same time, the security architecture allows that individual implementation defects break information hiding. In conclusion, this leads to thesis statement *TS1: The Java security architecture has design weaknesses that are commonly exploited by attackers.*

The results of our exploit study motivated us to investigate in depth the concept of information hiding and its role in the Java security architecture. Specifically, we showed that a large portion of the exploits in our sample set depend on illegal access to private members of system classes in order to implement their attacks. As we explain, the fundamental, underlying design issue with the Java security architecture that enabled many information-hiding attacks is a lack of *defense in depth*—multiple different security mechanisms must work together in order to prevent illegal member access, and hence in many cases a single vulnerability in any of these mechanisms is sufficient to break information hiding. To strengthen the platform against information-hiding attacks, we presented a *lightweight* mitigation strategy that combines a set of four countermeasures that our proof-of-concept implementation can integrate into even closed-source JREs. Our evaluation showed that this solution systematically blocks 84% of the information-hiding attacks contained in our exploit sample set, and we also explained how the remaining attacks can be blocked as well. We further showed that our solution is backward compatible and the runtime overhead induced by integrating the countermeasures is low—on average, there was a 2% decline in performance for the benchmarks in the DaCapo benchmark suite. In addition to this lightweight mitigation strategy, we further presented a *heavyweight* mitigation strategy. This alternative solution suggests a comprehensive redesign of the internals of the Java runtime. Specifically, we propose to consolidate the core of policy enforcement into a central, isolated component, the Java security monitor, which also provides a secure data store for sensitive field values. We further proposed extensions to the Java Language Specification and Java Virtual Machine Specification to prevent illegal access to private members of system classes, and ensure that system classes in the JCL store sensitive values in the security monitor's data store. We explained that implementing this mitigation strategy would require access to the JRE's source code and major engineering efforts. However, based the experience we gained by implementing and evaluating the lightweight mitigation strategy, we conclude that this heavyweight solution has the potential to fundamentally strengthen information hiding in the Java platform, and outperform our lightweight proof of concept in terms of both robustness and speed. In summary, this confirms thesis statement *TS2: The Java runtime can be transformed, in a backward-compatible manner, to defend against information-hiding attacks.*

Finally, we addressed an aspect of improper access control in the JRE that substantially decreased the maintainability of Java's codebase, while at the same time increasing its attack surface. Motivated by the findings of our exploit study, we showed that several security-sensitive methods in the JCL implement what we call *shortcuts*—instead of implementing thorough permission checks by consulting the security manager, they use hardcoded heuristics to determine whether a permission check can be skipped. By this, shortcuts implicitly elevate the privileges of certain callers, similar to privileged blocks. However, in contrast to the privileged block API, shortcuts elevate privileges implicitly which can easily lead developers to unknowingly introduce confused deputy vulnerabilities when extending or maintaining the codebase. We created a variant of the JCL that works almost without any implicit privilege elevation, whereby privileged blocks become the standard way for elevating privileges. As we explain, this substantially facilitates code maintenance, as well as automatic and manual program analysis. Also, certain attack vectors are blocked by this solution. The primary reason for implementing shortcuts originally was presumably performance optimization, as stack-based permission checks are comparatively expensive. However, using our new variant of the JCL we showed through a large-scale set of experiments that on modern hardware there is virtually no performance penalty when executing the DaCapo benchmark suite. We support this finding through a set of micro benchmark tests that confirm that absolute overheads induced through permission checks are in the order of microseconds. We finally assessed the impact of moving to a shortcut-free JCL for productive use by discussing usability and backward compatibility considerations, and also presented lessons learned that may serve as a guidance for the design and implementation of other complex security architectures. Hence, we confirmed thesis statement *TS3: The Java runtime's maintainability can be increased to lower the risk that refactorings introduce security vulnerabilities.*

6.2 Discussion

In this section, we put our work in context with other systems and programming languages, and also discuss general questions concerning secure software design.

6.2.1 Relevance to other systems

There are runtimes and programming languages similar to the JRE and Java. Hence, in the following, we discuss in detail the relevance of our findings to these technologies.

Microsoft .NET

Microsoft .NET was officially announced in the year 2000 [70], several years after Java has been presented to the public. The .NET framework is in many ways similar to the Java runtime—they both provide virtual machines for executing applications that were compiled to a platform-independent intermediate representation, for which they implement just-in-time compilation, automatic memory management, and various other features. Most interestingly in the context of this work is the fact that, originally, the .NET framework implemented "Code Access Security

(CAS)" [35, 18], an access control model very similar to the permission-based access control model in the Java security architecture. Both systems enforce configurable security policies by means of stack-based permission checks [9]. For this, all code is assigned a configurable set of permissions, and whenever access to sensitive resources is attempted, the runtime will inspect the call stack to determine if all code has been assigned sufficient privileges.

Despite these conceptual similarities between the Java platform and Microsoft .NET, there are important implementation-specific differences between the two systems. This includes in particular:

- Different instruction sets [86]—The .NET framework and Java runtime both execute applications that are deployed in platform-independent intermediate representations. The instruction set supported by the .NET framework is different to Java bytecode, and some of these differences may facilitate static verification of .NET applications when compared to Java-based applications. Paul et al. [86] illustrate this by the example of object creation. In Java, three instructions are used to instantiate a new object: `new` to allocate required heap space for the object, `dup` to duplicate the object reference of the new object on the stack, and `invokespecial` to invoke the newly created object's constructor. Verifying proper object initialization is thus especially difficult in methods that separate these three instructions by other instructions unrelated to object creation. As explained by Paul et al., improper object initialization in Java led to security vulnerabilities in the past. In .NET, object creation is implemented with a single instruction `newobj`, thus greatly facilitating verification and reducing the risk of security vulnerabilities in this context.
- Static permission checks—In contrast to Java, the .NET framework supports "declarative" security permissions [86, 67], which can be verified statically. This not only reduces the number of dynamic checks which potentially increases performance, it might also prevent attackers from bypassing certain checks at runtime.
- Strict policy enforcement—In Java, the security manager is disabled by default. Developers can implement their own custom security managers with alternative permission check procedures, and they can also disable policy enforcement at runtime, provided sufficient privileges. This allows for great flexibility, but as our study of Java exploits in Chapter 3 showed, this also increases the risk of security vulnerabilities and enables attack strategies that can be used to disable policy enforcement. In .NET, policy enforcement follows stricter rules and is less flexible [86]. Until version 3.5 of the framework, policy enforcement was enabled by default [71]. Developers have no means to customize permission checking procedures, besides adjusting security policies.
- Extended stack inspection modification [86]—In Java, stack inspection is triggered by a call to the security manager. Methods can temporarily elevate their privileges by using `doPrivileged()` in order to modify the stack inspection routine, see Section 2.2.1. In .NET, the means to modify stack inspection were extended by additional methods. The equivalent to `doPrivileged()` in Java is `Assert` in .NET. In addition, .NET implements `Deny`, which temporarily denies certain permissions to callers of the method that invokes `Deny`, and `PermitOnly`, which enforces that callers of the method that invokes `PermitOnly` can only use specified permissions [35, p. 111], even though the security policy in effect may grant more permissions to the concerning code. In abstract terms, this

means that Java only provides means to *elevate* privileges of code, while .NET also provides additional means to *restrict* the privileges of code. Proper use of these mechanisms might lower the risk that trusted code can be abused as a confused deputy by attackers, or accidentally expose sensitive resources.
For example, a trusted method in the framework responsible for displaying user interface components might decide to use `PermitOnly` to refrain from all privileges except for the `UIPermission` [8]. In effect, even if this method and its callees implement, e.g., a confused deputy vulnerability due to an insecure use of reflection, an attacker is limited when exploiting this vulnerability due to the restricted set of permissions in this context. In this example, `PermitOnly` allows that the framework can generally be assigned all privileges, but still restrict itself for certain use cases to follow the principle of least privilege. Similarly, computation-heavy framework methods implementing cryptographic primitives or media processing might use `Deny` to refrain from all privileges, in case they do not require direct access to sensitive resources of the host.

- Finally, we showed that restricted classes in Java significantly increase the attack surface as they refrain from proper permission checks. We are not aware of any restricted classes in the .NET framework, or any similar design flaws.

The above list of differences illustrates that .NET's security design already addresses many shortcomings of Java's security architecture, which precludes us from simply transferring the problem descriptions and solution strategies presented in this work to the .NET platform. In fact, Java vendors should consider porting some of the concepts implemented in .NET to the Java platform to enhance security, provided that this can be done in a backward-compatible way.

However, despite these many differences, there is a concept that we extensively discussed in this work as it significantly contributes to Java's insecurity, and we find that the .NET platform implements a very similar concept. In Chapter 5, we explained that several sensitive methods in the JCL implement hardcoded shortcuts, which skip permission checks for performance reasons if certain constraints on the call stack are satisfied. We showed that these shortcuts significantly decrease the Java platform's maintainability and attack resistance. As we find, the .NET platform implements a very similar concept with `LinkDemand`. In the general case, permission checks in .NET are triggered with `Demand`, which results in stack inspection. As Freeman et al. explain [35, p. 133], "[a] `LinkDemand` is similar to a `Demand`, but it is evaluated only when a caller first links to your code and only checks the permissions of the immediate caller as opposed to the permissions of all callers on the call stack. `LinkDemand` provides a lightweight alternative to the `Demand` and is used to reduce the performance hit of performing stack walks on frequently called and time-critical methods. However, because the permissions of all callers are not checked, `LinkDemand` leaves your code susceptible to attacks". Although CAS in .NET is implemented differently than access control in Java, there are remarkable conceptual similarities between shortcuts in the JCL and `LinkDemand` in .NET. From this we conclude that the discussions and findings presented in Chapter 5 potentially also apply to the .NET platform. However, this is not necessarily the case, and more work is needed to clarify this.

Finally, it is important to note that Microsoft decided to move away from CAS. Specifically, they state that CAS will no longer be supported as a means to separate untrusted code from trusted code, and they explicitly advise against loading or executing untrusted code unless other

security measures are used to prevent harm [67]. Consequently, CAS is no longer enabled by default in current versions of the framework. The reasons for the abandonment of this access control model are manifold [69]. This includes that different versions of .NET implement access control in different ways, which is especially problematic on machines that run multiple different versions [68]. Also, it is difficult to manage CAS policy updates in enterprise settings. Further, managed applications and native applications behave differently, "often in confusing and undesirable ways" [69]. Finally, the model is considered complex and hard to use.

Instead, Microsoft developed alternative security mechanisms. For example, current versions of Windows provide AppContainer isolation [66], an approach to sandboxing applications on the operating system level, by using process isolation and restricted access to sensitive interfaces, such as file and network access. AppContainer can be used to restrict the execution of Microsoft Store or Universal Windows apps [65], and also Microsoft Edge makes use of this concept to defend against remote code execution attacks [72].

Android

The Java platform and language were originally released in the mid-1990's, but the technological landscape has evolved since then. Nowadays, mobile consumer devices such as smartphones and tablets are widely distributed, and most of these devices are based on the Android platform [53]. In the context of this work, it is specifically interesting that the Android system is primarily implemented in Java. Also, Java is the primary language for implementing Android apps [104]. The Linux kernel is the technological foundation of the Android platform, on top of which it implements a runtime environment for Java-based apps ("Android Runtime (ART)"), as well as a set of core and native libraries [42]. On top of these libraries, the Android platform further implements the "Java API Framework", which provides all available features of the platform and the device to app developers [42].

From a security perspective, the Java code that belongs to the Android platform itself can be seen as *trusted code*, similar to the JCL code in standard JRE installations. Also, as apps are developed by potentially untrusted third parties, app code retrieved from external sources such as app stores can be seen as *untrusted code*. Hence, for security and privacy reasons, there is a need to isolate apps from one another, restrict access to hardware interfaces and storage, and prevent tampering with the operating system's internals.

Since the Android platform itself and the apps it executes are primarily written in Java, and the security goals are comparable to desktop and server environments, one may assume that the Android platform implements the same security model as the Java standard editions, such as the J2SE. However, Android's security model is fundamentally different to the security model of standard Java editions studied in this work. Most noticeable is the fact that Android does not use stack-based permission checks to implement access control. As the Android API reference states, the `SecurityManager` class is legacy code that is not supposed to be used [41], and further explains: "Secure managers do *not* provide a secure environment for executing untrusted code and are unsupported on Android. Untrusted code cannot be safely isolated within a single VM on Android" [41]. Instead, the Android platform executes all apps in individual Linux processes, and each of these processes runs under an app-specific UID, with access to only app-specific file directories [31]. Permissions are assigned to entire apps, rather than to classes or

protection domains individually as in the JRE's security model. Hence, access-control decisions in Android are performed per-process and permission enforcement is in many cases implemented on the Linux kernel level [31]. In comparison to standard Java editions, the advantage of such a security design is that even native code shipped with otherwise Java-based apps is subject to policy enforcement, as also the native code is executed within the same process that runs the Java-based components of the app.

The disadvantage of this model, however, is the fact that policy enforcement is more coarse-grained on Android than on standard Java editions—permissions can only be assigned on the level of entire apps, but there is no means for developers or users to restrict only certain parts of an app. In practice, this is an issue, because many apps include potentially untrusted third-party code, e.g., libraries for advertisement, which can result in security and privacy issues even in otherwise trustworthy apps.

Due to these fundamental differences between the security architecture implemented in Android and the security architecture of standard Java editions, the technical details of Java's weaknesses revealed and discussed in this work cannot simply be entirely transferred to the Android platform. However, there are specific technical aspects that concern both systems. Peles et al. [87] showed that vulnerabilities in object serialization and deserialization are a serious threat to Android, a finding that we also encountered and discussed in the exploit study presented in Chapter 3. There might be solutions to systematically strengthen the concept of serialization and deserialization in such a way that flaws in Android and standard Java editions are uniformly addressed, however, further research is needed.

Rust and WebAssembly

The Rust programming language is a relatively new development that aims for memory-safe, low-level code. By design, it avoids a broad range of memory-related programming defects through compile-time checks, including buffer and stack overflows, as well as illegitimate accesses to uninitialized or deallocated buffers [64].

WebAssembly is a low-level, and yet platform-independent intermediate representation for code to be executed in a web browser [47]. It can serve as a compilation target for code written in Rust, but also other languages, like C++ [75]. It is intended to complement, and not replace, JavaScript, to allow that performance-demanding applications can be easily developed and ported to the web.

The combination of Rust and WebAssembly is thus addressing a use case for which Java applets were originally designed. But, there are fundamental differences between the two solutions. Rust and WebAssembly implement an entirely different memory model than the JVM, and they do not implement a policy enforcement mechanism similar to the Java security architecture, which is why the design weaknesses and solutions discussed in this work are generally not applicable here. However, especially low-level implementation defects can undermine the security guarantees provided by WebAssembly, similar to what has previously been observed for the JVM. For example, in 2017 a type confusion vulnerability in Google Chrome's implementation of WebAssembly was filed under CVE-2017-15413, which allowed for remote attacks through crafted web pages. We showed in Chapter 3 and 4 that type confusion is also a type of vulnerability that attackers commonly use to compromise the JRE. Future research is needed

to investigate how these low-level implementation defects can be addressed in browsers, and whether there can be a unified solution that strengthens both WebAssembly and Java.

6.2.2 Secure software design[1]

Our study of the Java security architecture presented in this work can be seen as a case study that raises general questions concerning the secure design of complex software systems. Despite significant efforts, many large-scale systems were repeatedly exposed to single vulnerabilities which undermined their entire security architectures. Such fragility must be inherent to the implemented architectures, and continuous efforts in patching individual bugs does not solve the underlying problems in system design.

We can measure the brittleness of software in two dimensions:

> **Compile-time metric**—What is the minimum number of lines in the target system's source code that need to be modified to compromise the whole system? In the case of Java, a single line of code is sufficient. Inserting `return null;` in the beginning of `System.getSecurityManager()` is just one example, and there are many alternatives.
>
> **Runtime metric**—What is the minimum number of bits in memory that need to be modified to compromise the target system? For Java, the number of bits is lower or equal to the size of a single object reference in memory. Flipping the bits that represent `System.security` is just one example.

Low numbers for any of the two metrics represent high brittleness and may indicate fundamental flaws in system design. Besides the obvious examples that we gave to illustrate Java's brittleness, there is an unknown number of non-obvious modifications that would equally compromise policy enforcement. In fact, replacing a single occurrence of == by = might break a large number of highly security-critical software systems. High brittleness not only implies that a software system is susceptible to security bugs introduced unintentionally, it also implies a high potential for hiding hard-to-find backdoors.

Java was designed as a self-protecting system, i.e., the security mechanisms that are supposed to protect the system are part of the system itself. This inherently leads to circular dependencies, which presumably contributes to brittleness. An alternative to a self-protecting system is a system that is compartmentalized such that access control decisions are carried out by a dedicated component, the *reference monitor*. This fundamental concept has its origins in the design of secure military systems, and it aims for reliable and hard-to-bypass policy enforcement. Early work in this area [7] defines a set of principles, according which the reference monitor has to be tamper proof, be involved in all security-related decisions, and simple enough to be verifiable. The similar concept of a *security kernel* directly associates tamper resistance with isolation [96]. The redesign of Java's security architecture that we proposed in Section 4.5 is fully in line with these principles. Previous work referred to the `SecurityManager` as an implementation of a reference monitor [27], however, we argue that it does not sufficiently comply to any of the above principles.

[1] This section was taken with minor modifications from [50]

Various approaches to bug detection have been proposed in the past, but the fragility of complex software is, to large extents, caused by shortcomings in its security design. Our study of the Java security architecture can be seen as a case study that supports this claim. Further research is needed to provide a better understanding of how to detect and prevent security design flaws.

6.3 Directions for future work

In addition to the topics for future research already discussed in the previous section, we want to highlight two topics of particular interest.

Java Module System—In this work, we focused on Java 6, 7 and 8. More current versions of Java that were recently released to the public implement an entirely new modularization model, the Java Module System (JMS) [81]. The JMS has changed the way that Java-based applications are developed, deployed and executed, but to the best of our knowledge, a large-scale analysis on its impact on Java platform security has not been independently carried out yet. Future work should thus focus on a detailed analysis of this new model and its impact on the Java security architecture. In particular, it should be evaluated to which extent the design weaknesses outlined in this work have been addressed by the changes, and whether the JMS can be leveraged to implement alternative mitigation strategies.

Operating-system-supported isolation—From the beginning, the Java runtime was designed to contain untrusted code within the same operating system process that contained trusted code. In other words, Java implements *intra-process isolation*, and the Java security architecture is mostly concerned with addressing the risks and problems naturally associated with this approach to code isolation. What we observe, however, is a rise of more fundamental isolation strategies, leveraging operating system features such as process separation and kernel-level access control. Examples for this were discussed above and include Microsoft's implementation of AppContainer, as well as Android's approach to app isolation on the basis of individual Linux processes. Another example is the Firefox browser, which uses individual child processes to isolate web contents of different origins for enhanced security and stability [74], a concept also applied by other browsers, such as Chrome [44]. Future work should evaluate whether process separation and possibly advanced techniques like AppContainer can be used in the JRE to achieve the same security goals that were originally pursued by the Java security architecture in a backward-compatible way, with the goal of achieving more reliable and secure code isolation in Java.

Bibliography

[1] OpenJDK. http://openjdk.java.net/.

[2] Scopus. https://www.scopus.com.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[5] M. Abadi and C. Fournet. Access control based on execution history. In *NDSS*, volume 3, pages 107–121, 2003.

[6] Adobe. Document management—Portable document format—Part 1: PDF 1.7 (PDF 32000-1:2008). Standard, 2008.

[7] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, ANDERSON (JAMES P) AND CO FORT WASHINGTON PA, 1972.

[8] F. Balena. *Programming Microsoft Visual Basic. NET Version 2003*. Antonio Faustino, 2004.

[9] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of functional programming*, 15(2):131–177, 2005.

[10] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54:69–80, 2001.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java

benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[13] Bundesamt für Sicherheit in der Informationstechnik. Java Sicherheitsempfehlungen. https://www.bsi-fuer-buerger.de/BSIFB/DE/Empfehlungen/EinrichtungSoftware/EinrichtungBrowser/Sicherheitsmassnahmen/Java/Java_Sicherheistempfehlungen/java_sicherheitsempfehlungen_node.html. Online; accessed 23-Aug-2018.

[14] B.-M. Chang. Static check analysis for java stack inspection. *ACM SIGPLAN Notices*, 41(3):40–48, 2006.

[15] P. Chen, R. Wu, and B. Mao. Jitsafe: a framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292, 2013.

[16] S. Chiba. Javassist—a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, volume 174, page 21, 1998.

[17] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, et al. Cheri jni: Sinking the java security model into the c. *ACM SIGOPS Operating Systems Review*, 51(2):569–583, 2017.

[18] C. Chu. Introduction to microsoft. net security. *IEEE Security & Privacy*, 6(6):73–78, 2008.

[19] C. Cifuentes, A. Gross, and N. Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 7–12. ACM, 2015.

[20] Cisco. 2013 Cisco Annual Security Report, 2013.

[21] Cisco. 2014 Cisco Annual Security Report. https://www.cisco.com/c/dam/assets/global/UK/pdfs/executive_security/sc-01_casr2014_cte_liq_en.pdf, 2014.

[22] Z. Coker, M. Maass, T. Ding, C. Le Goues, and J. Sunshine. Evaluating the flexibility of the java sandbox. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 1–10. ACM, 2015.

[23] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[24] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[25] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[26] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *USENIX Security Symposium*, pages 395–410, 2008.

[27] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From Hotjava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Conference on Security and Privacy*, SP'96, pages 190–200, Washington, DC, USA, 1996. IEEE Computer Society.

[28] D. Dean and D. S. Wallach. Security flaws in the hotjava web browser. *Department of Computer Science, Princeton University*, 1995.

[29] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the. net platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[30] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106. Springer, 2009.

[31] N. Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

[32] FireEye. Brewing up trouble: Analyzing four widely exploited java vulnerabilities. `https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-java-vulnerabilities.pdf`. Online; accessed 09-Dec-2018.

[33] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *2009 Annual Computer Security Applications Conference*, pages 363–372. IEEE, 2009.

[34] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *ACM SIGPLAN Notices*, volume 37, pages 307–318. ACM, 2002.

[35] A. Freeman and A. Jones. *Programming. NET Security: Writing Secure Applications Using C# or Visual Basic. NET*. " O'Reilly Media, Inc.", 2003.

[36] J. Y. Gil, K. Lenz, and Y. Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 297–308. ACM, 2011.

[37] L. Gong. Secure java class loading. *IEEE Internet Computing*, 2(6):56–61, 1998.

[38] L. Gong. Java security: a ten year retrospective. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 395–405. IEEE, 2009.

[39] L. Gong and G. Ellison. *Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[40] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[41] Google. Android Platform API Reference SecurityManager. `https://developer.android.com/reference/java/lang/SecurityManager`. Online; accessed 07-Apr-2019.

[42] Google. Android Platform Architecture. `https://developer.android.com/guide/platform`. Online; accessed 07-Apr-2019.

[43] Google. Android Security Bulletin—July 2018. `https://source.android.com/security/bulletin/2018-07-01`. Online; accessed 24-Apr-2019.

[44] Google. Mitigating Spectre with Site Isolation in Chrome. `https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html`. Online; accessed 19-May-2019.

[45] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 402–416. IEEE, 2008.

[46] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of java virtual machines. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121. ACM, 2006.

[47] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[48] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[49] A. Herzog and N. Shahmehri. Performance of the java security manager. *Computers & Security*, 24(3):192–207, 2005.

[50] P. Holzinger and E. Bodden. A Systematic Hardening of Java's Information Hiding. To be published.

[51] P. Holzinger, B. Hermann, J. Lerch, E. Bodden, and M. Mezini. Hardening Java's Access Control by Abolishing Implicit Privilege Elevation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1027–1040, May 2017.

[52] P. Holzinger, S. Triller, A. Bartel, and E. Bodden. An in-depth study of more than ten years of Java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 779–790. ACM, 2016.

[53] IDC. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/os. Online; accessed 07-Apr-2019.

[54] IEEE. IEEE John von Neumann Medal Recipients. https://www.ieee.org/about/awards/bios/vonneumann-recipients.html. Online; accessed 22-Aug-2018.

[55] I. Ion, B. Dragovic, and B. Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 233–242. Ieee, 2007.

[56] Kaspersky Lab. Java under attack – the evolution of exploits in 2012-2013. https://securelist.com/kaspersky-lab-report-java-under-attack/57888/. Online; accessed 09-Dec-2018.

[57] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[58] S. Koivu. Java trusted method chaining (cve-2010-0840/zdi-10-056). http://slightlyrandombrokenthoughts.blogspot.com/2010/04/java-trusted-method-chaining-cve-2010.html, 2010. Online; accessed 22-Nov-2018.

[59] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *ACM Sigplan Notices*, volume 37, pages 359–372. ACM, 2002.

[60] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[61] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. 2001.

[62] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification - Java SE 8 Edition*. 2015.

[63] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: the java unsafe api in the wild. In *ACM Sigplan Notices*, volume 50, pages 695–710. ACM, 2015.

[64] N. D. Matsakis and F. S. Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[65] Microsoft. /APPCONTAINER. https://docs.microsoft.com/de-de/cpp/build/reference/appcontainer?view=vs-2019. Online; accessed 14-Apr-2019.

[66] Microsoft. AppContainer Isolation. https://docs.microsoft.com/en-us/windows/desktop/secauthz/appcontainer-isolation. Online; accessed 13-Apr-2019.

[67] Microsoft. Code Access Security Basics. https://docs.microsoft.com/de-de/dotnet/framework/misc/code-access-security-basics. Online; accessed 14-Apr-2019.

[68] Microsoft. Every CLR has Independent CAS Policy. https://blogs.msdn.microsoft.com/shawnfa/2006/07/11/every-clr-has-independent-cas-policy/. Online; accessed 14-Apr-2019.

[69] Microsoft. Is CAS dead in .NET 4? https://blogs.msdn.microsoft.com/shawnfa/2010/02/24/is-cas-dead-in-net-4/. Online; accessed 14-Apr-2019.

[70] Microsoft. Microsoft Unveils Vision for Next Generation Internet. https://news.microsoft.com/2000/06/22/microsoft-unveils-vision-for-next-generation-internet/. Online; accessed 13-Apr-2019.

[71] Microsoft. NetFx40_LegacySecurityPolicy Element. https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/runtime/netfx40-legacysecuritypolicy-element. Online; accessed 14-Apr-2019.

[72] Microsoft. Strengthening the Microsoft Edge Sandbox. https://blogs.windows.com/msedgedev/2017/03/23/strengthening-microsoft-edge-sandbox/. Online; accessed 14-Apr-2019.

[73] Microsoft. Microsoft Security Advisory 4022344. https://docs.microsoft.com/en-us/security-updates/securityadvisories/2017/4022344, 2017. Online; accessed 21-Aug-2018.

[74] Mozilla. Multiprocess Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox. Online; accessed 14-Apr-2019.

[75] Mozilla. WebAssembly Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. Online; accessed 14-Apr-2019.

[76] Mozilla. Mozilla Foundation Security Advisory 2014-29. https://www.mozilla.org/en-US/security/advisories/mfsa2014-29/, 2014. Online; accessed 17-Aug-2018.

[77] MWR Labs. JavaScript Privilege Escalation in Adobe Reader. https://labs.mwrinfosecurity.com/assets/1008/original/mwri_advisory_javascript_privilege_escalation_in_adobe_reader.pdf, 2015. Online; accessed 17-Aug-2018.

[78] J. W. Oh. Recent java exploitation trends and malware. https://media.blackhat.com/bh-us-12/Briefings/Oh/BH_US_12_Oh_Recent_Java_Exploitation_Trends_and_Malware_WP.pdf. Online; accessed 09-Dec-2018.

[79] Oracle. Go Java. https://go.java/index.html. Online; accessed 23-Aug-2018.

[80] Oracle. Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/. Online; accessed 22-Aug-2018.

[81] Oracle. JEP 261: Module System. https://openjdk.java.net/jeps/261. Online; accessed 14-Apr-2019.

[82] Oracle. Secure coding guidelines for java se. https://www.oracle.com/technetwork/java/seccodeguide-139067.html. Online; accessed 16-Dec-2018.

[83] Oracle. Migrating from java applets to plugin-free java technologies. https://www.oracle.com/technetwork/java/javase/migratingfromapplets-2872444.pdf, January 2016.

[84] Oracle. Text Form of Oracle Critical Patch Update - January 2018 Risk Matrices. http://www.oracle.com/technetwork/security-advisory/cpujan2018verbose-3236630.html, 2018. Online; accessed 21-Aug-2018.

[85] T. Ormandy. MsMpEng: Remotely Exploitable Type Confusion in Windows 8, 8.1, 10, Windows Server, SCEP, Microsoft Security Essentials, and more. https://bugs.chromium.org/p/project-zero/issues/detail?id=1252&desc=5, 2017. Online; accessed 21-Aug-2018.

[86] N. Paul and D. Evans. .net security: lessons learned and missed from java. In *20th Annual Computer Security Applications Conference*, pages 272–281. IEEE, 2004.

[87] O. Peles and R. Hay. One class to rule them all 0-day deserialization vulnerabilities in android. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, pages 5–5. USENIX Association, 2015.

[88] C. P. Pfleeger, S. L. Pfleeger, and J. Margulies. Security in computing. 2015.

[89] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 149–163. IEEE, 2007.

[90] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *European Conference on Object-Oriented Programming*, pages 362–386. Springer, 2005.

[91] R. Pompon. Assume breach. In *IT Security Risk Control Management*, pages 13–21. Springer, 2016.

[92] RedHat. Bug 1523129 - (CVE-2017-15413) CVE-2017-15413 chromium-browser: type confusion in webassembly. https://bugzilla.redhat.com/show_bug.cgi?id=1523129, 2017. Online; accessed 17-Aug-2018.

[93] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[94] J. Rose, C. Thalinger, and M. Chung. JEP 176: Mechanical checking of caller-sensitive methods. `https://openjdk.java.net/jeps/176`. Online; accessed 08-Dec-2018.

[95] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[96] R. R. Schell, P. J. Downey, and G. J. Popek. Preliminary notes on the design of secure military computer systems. Technical report, ELECTRONIC SYSTEMS DIV HANSCOM AFB MA, 1973.

[97] J. Schlumberger, C. Kruegel, and G. Vigna. Jarhead analysis and detection of malicious java applets. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 249–257. ACM, 2012.

[98] Security Explorations. Security vulnerability notice - se-2012-01-ibm-2. `http://www.security-explorations.com/materials/SE-2012-01-IBM-2.pdf`.

[99] Security Explorations. Security vulnerability notice - se-2012-01-ibm-5. `http://www.security-explorations.com/materials/SE-2012-01-IBM-5.pdf`.

[100] Security Explorations. Security vulnerability notice - se-2012-01-oracle-14. `http://www.security-explorations.com/materials/SE-2012-01-ORACLE-14.pdf`.

[101] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[102] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. *ACM SIGPLAN Notices*, 46(6):343–354, 2011.

[103] N. Stojanovski, M. Gusev, D. Gligoroski, and S. J. Knapskog. Bypassing data execution prevention on microsoftwindows xp sp2. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 1222–1226. IEEE, 2007.

[104] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.

[105] TIOBE Software BV. TIOBE Index. `https://www.tiobe.com/tiobe-index/`. Online; accessed 22-Aug-2018.

[106] R. Toledo, A. Nunez, E. Tanter, and J. Noye. Aspectizing java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, 2012.

[107] S. Türpe. Idea: Usable platforms for secure programming–mining unix for insight and guidelines. In *International Symposium on Engineering Secure Software and Systems*, pages 207–215. Springer, 2016.

[108] D. S. Wallach, A. W. Appel, and E. W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):341–378, 2000.

[109] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[110] C. Wressnegger and K. Rieck. Looking back on three years of flash-based malware. In *Proceedings of the 10th European Workshop on Systems Security*, page 6. ACM, 2017.

[111] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Comprehensive analysis and detection of flash-based malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 101–121. Springer, 2016.

[112] K.-P. Yee. User interaction design for secure systems. In *International Conference on Information and Communications Security*, pages 278–290. Springer, 2002.

[113] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

A

Extension to the JVM instruction set

Instruction: **getfield_critical**

Operation:
Fetch critical field from object.

Format:

getfield_critical indexbyte1 indexbyte2

Operand stack:

..., objectref → ..., value

See also:
getfield

Instruction: **getstatic_critical**

Operation:
Get critical static field from class.

Format:

getstatic_critical indexbyte1 indexbyte2

Operand stack:

..., → ..., value

See also:
getstatic

Instruction: **putfield_critical**

Operation:
Set critical field in object.

Format:

putfield_critical indexbyte1 indexbyte2

Operand stack:

..., objectref, value → ...

See also:
putfield

Instruction: **putstatic_critical**

Operation:
Set critical static field in class.

Format:

putstatic_critical indexbyte1 indexbyte2

Operand stack:

..., value → ...

See also:
putstatic

Instruction: **invokespecial_critical**

Operation:
Invoke private instance method.

Format:

invokespecial_critical indexbyte1 indexbyte2

Operand stack:

..., objectref, [arg1, [arg2 ...]] → ...

See also:
invokespecial

Instruction: **invokestatic_critical**

Operation:
Invoke private static method.

Format:

invokestatic_critical indexbyte1 indexbyte2

Operand stack:

..., [arg1, [arg2 ...]] → ...

See also:
invokestatic