An In-Depth Study of More Than Ten Years of Java Exploitation

Philipp Holzinger[1], Stefan Triller[1], Alexandre Bartel[2], and Eric Bodden[3,4]
[1]Fraunhofer SIT, [2]Technische Universität Darmstadt, [3]Paderborn University, [4]Fraunhofer IEM
[1]{firstname.lastname}@sit.fraunhofer.de, [2]alexandre.bartel@cased.de
[3]eric.bodden@uni-paderborn.de

ABSTRACT

When created, the Java platform was among the first runtimes designed with security in mind. Yet, numerous Java versions were shown to contain far-reaching vulnerabilities, permitting denial-of-service attacks or even worse allowing intruders to bypass the runtime's sandbox mechanisms, opening the host system up to many kinds of further attacks.

This paper presents a systematic in-depth study of 87 publicly available Java exploits found in the wild. By collecting, minimizing and categorizing those exploits, we identify their commonalities and root causes, with the goal of determining the weak spots in the Java security architecture and possible countermeasures.

Our findings reveal that the exploits heavily rely on a set of nine weaknesses, including unauthorized use of restricted classes and confused deputies in combination with caller-sensitive methods. We further show that all attack vectors implemented by the exploits belong to one of three categories: single-step attacks, restricted-class attacks, and information hiding attacks.

The analysis allows us to propose ideas for improving the security architecture to spawn further research in this area.

1. INTRODUCTION

From a security point of view, a virtual machine's goal is to contain the execution of code originating from untrusted sources in such a way that it cannot impede the security goals of the host machine. For instance, the code should not be able to access sensitive information to which access has not been granted, nor should it be able to launch a denial-of-service attack. Many virtual machines try to contain untrusted code through a so-called sandbox model. Conceptually, a sandbox runs the untrusted code in a controlled environment, by separating its execution and its data from that of trusted code, and by allowing it only to have access to a limited and well-defined set of system resources.

This paper investigates more than ten years of insecurities and exploitation of the Java platform, whose security concepts rely heavily on such sandbox model. Conceptually, the Java Runtime Environment (JRE) uses a sandbox to contain code whose origin is untrusted in a restricted environment. When executing a Java applet from an untrusted site within a browser, its access is controlled. Sandboxed applets are only allowed to perform a very limited set of tasks such as making network connections to the host they were loaded from, or display HTML documents.[1] A second use case of sandboxing in Java is on the server side: application servers use the sandbox mechanisms to isolate from one another and from the host systems the applications they serve.

While conceptually easy to grasp, the Java sandbox is actually anything but a simple box. Instead it comprises one of the world's most complex security models in which more than a dozen dedicated security mechanisms come together to—hopefully—achieve the desired isolation. To just give some examples: bytecode verification must prevent invalid code from coming to execution, access control must correctly tell apart trusted from untrusted code, and to prevent the forging of pointers type checking must in all cases properly distinguish pointer references from numeric values. As a consequence, the "sandbox" is only as good as the joint security architecture and implementation of all those different mechanisms that comprise the sandbox. Adding to that, the code implementing the sandbox has evolved over far more than a decade, involving dozens of developers, with virtually none of the original creators remaining in charge today, and with the lead maintenance of Java moving from Sun Microsystems to Oracle Inc. When considering all this, it may come as less of a surprise that over the years the Java runtime has seen a large number of devastating vulnerabilities and attacks, most of which lead to a full system compromise: an attacker would be able to inject and execute any code she desires, at the very least with the operating-system privileges assigned to the user running the Java virtual machine [7, 8]. Security vulnerabilities are present in different parts of the complex sandbox mechanism, involving issues such as type confusion, deserialization issues, trusted method chaining or confused deputies.

With Java being a runtime deployed on literally billions of devices, it is one of the most prevalent software systems in use today. Hence naturally, Java vendors such as Oracle and IBM are eager to fix vulnerabilities once they become known. But over the past years, the crafting of exploits by attackers and the crafting of patches by vendors has become a continuing arms race. Oftentimes, security patches lit-

[1]https://docs.oracle.com/javase/tutorial/deployment/applet/security.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'16, October 24 - 28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4139-4/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2976749.2978361

erally only “patch” the discovered hole without addressing the actual underlying security problem. In many cases this has allowed attackers to replace one patched part of an exploit by another one based on a newly found, similar root cause of the vulnerability. In other cases, the fragmentation of the platform caused additional problems. For instance, the exploit for CVE-2013-5838 impacting Oracle Java 7 update 25 still works with minor modification on Oracle Java 8 update 74 [7]. Similarly, the exploit for CVE-2013-5456 against IBM Java SDK 7.0.0 before SR6, still works with minor modifications against IBM SDK 7.1 [8]. In result, it seems as if even the Java vendors have lost track of the sandbox mechanisms' original security goals, the interaction protocols between those mechanisms and how the security goals of those individual mechanisms and their interactions are actually meant to be enforced within Java. One of the goals of this paper is to bring back to light some of that crucial knowledge, by highlighting the inner workings of past and current exploits and the vulnerabilities and weaknesses to which they relate.

In this work we thus present an overdue in-depth study of all publicly available Java exploits we were able to find. We harvested 87 different exploits from the Internet and reduced each of them to a minimal form, retaining only the code crucial to achieving the goal of the exploit (full bypass, DoS, etc.). Each exploit was validated to fulfil its goal on the Java versions it targets. From the minimal representation, we next manually split every exploit into independent steps that we call *primitives*. This allows us to reason about the different steps an exploit needs to perform to reach the attacker's goal, at a level higher than the code. By comparing primitives, we are able to compare and cluster the behavior of exploits. This clearly points us to weak spots of the Java platform which are used in many different attacks.

To summarize, this paper makes the following contributions:

- a collection of 61 working Java exploits, based on a set of 87 original exploits,
- an analysis and categorization of the Java exploits in terms of intended behavior and primitives,
- an analysis of Java in terms of its weak spots with respect to security, and
- potential security fixes for those weak spots.

We make the full documentation of the exploit sample set publicly available along with this paper[2].

The remainder of the paper is organized as follows. In Section 2, we introduce the reader to the basics of the Java security model. Next, in Section 3, we detail how we collected the set of Java exploits. We describe how we model the exploits in Section 4 and explain our findings in Section 5. Finally, we discuss the related work in Section 6 and conclude in Section 7.

2. BACKGROUND

In this section, we provide a basic introduction to the Java security model. Additionally, we will present a number of features that do not belong to the core implementation of the security model, but that will be heavily involved in the discussions of this paper. We limit ourselves to the crucial parts that are required to understand the following exploit analysis.

2.1 The Java security model

First we will introduce some of the essential mechanisms of the Java security model. Gong and Ellison provide a more comprehensive documentation [9].

Classloading.

Before Java applications can be executed, they need to be loaded into the runtime. For this, the Java platform provides a set of classloaders. Both the applications and the runtime itself use these classloaders to dynamically load new code from various sources, e.g., from the local file system, or network resources. During initialization of the Java runtime, the Java Virtual Machine (JVM) uses a bootstrap classloader to load required parts of the Java Class Library (JCL) into memory. The JCL contains all the classes that implement Java's standard API, such as `java.lang.Object`, or `java.lang.Class`. In the following, we will call such classes *system classes*. The JVM loads application classes with another classloader instance, the *application classloader*. The process that converts a byte representation of a class into an instance of `java.lang.Class` is known as *class definition*. Each newly defined class is assigned a protection domain, which itself is associated with a specific set of permissions. Classes that were loaded by the bootstrap classloader, including all system classes, are trusted classes, and thus associated with a protection domain that provides all permissions. Application classes are, by default, untrusted and thus assigned a protection domain with limited permissions.

For security reasons, some parts of the JCL are off limits. These parts reside in specific packages, known as *restricted packages* containing *restricted classes*. Such classes cannot be loaded by application classes unless those were explicitly given permission to do so. Examples for restricted packages are `sun.**`, or `com.sun.imageio.**`. Well-known restricted classes are `sun.misc.Unsafe` and `sun.awt.SunToolkit`. If an attacker manages to invoke functionality of restricted packages, this is usually sufficient for a full bypass of all security features. We found that the number of restricted packages increased significantly over time. Java 1.7.0 contained four restricted packages (not counting subpackages), version 1.7.0u11 contained eight, but the current version 1.8.0u92 contains 47 restricted packages.

Stack-based access control.

The JCL provides sensitive functionality, such as file and network access. All those features are guarded by a permission check, triggered through a call to `SecurityManager.check*`. Permission checks are implemented by means of stack-based access control. Whenever a check has been triggered, the runtime will inspect the call stack to check whether all calling classes are associated with protection domains that own the requested permission. If any of the callers does not have permission to use the desired functionality, an exception is thrown to deny access to the functionality.

There is one way to diverge from this basic algorithm. By using `AccessController.doPrivileged`, a system class can vouch for the fact that the specific way in which the sensitive functionality is used is safe to be used also by untrusted code. Whenever a permission check has been trig-

[2]https://github.com/pholzinger/exploitstudy

gered, stack inspection will stop at the first caller that has called `doPrivileged`, and check only permissions of callers up until there.

Java applications can be run with and without an active `SecurityManager`. If no `SecurityManager` has been set, this bypasses all calls to `SecurityManager.check*`, thus offering unlimited access to features of the JRE. If, however, a `SecurityManager` is in charge, the instance that is responsible for access control is stored in the private field `java.lang.System.security`. Naturally, if an attacker succeeds in setting this field to `null`, this results in a full security bypass.

Information hiding.

Information hiding, while often not perceived as a security feature, is in fact crucial to the security of the entire platform. Access to private fields and methods of system classes, such as `java.lang.System.security`, allow an attacker to bypass all security checks. Vulnerabilities that can be used to circumvent visibility rules are thus highly critical.

Type safety.

Type safety is the last crucial aspect of the Java security model that we want to highlight. At any time of program execution, it is important for the runtime to keep track of the types of objects, and the operations that one may perform on them. An attacker can use vulnerabilities to create *type confusion*, thus allowing her to perform an action on an object of type A, while the Java runtime thinks the action is performed on type B. As an example, the attacker can set the field `CustomClass.security` to *null*, which is not a security breach under normal conditions. If, however, `CustomClass.security` is actually `java.lang.System.security`—and the JRE will allow this action only because of a type confusion—an attacker can use this to disable the `SecurityManager`. This is why type safety is an essential aspect of the Java security model.

2.2 Special features

The JRE provides a set of features that we consider special in the context of this analysis because they conceptually work against the security model.

One of those features is the reflection API. Using reflection, executed classes can inspect themselves, other classes, or the call stack, among other things. On the one hand, this is a valuable feature, which is used to implement, e.g., testing frameworks, debuggers, and development environments [6]. On the other hand, however, reflection can be used to bypass information hiding. If given permission, a class can use the reflection API to modify private field values of other classes, thus violating visibility rules. Certain system classes use reflection to implement *caller-sensitive* methods. Such methods inspect the call stack and vary their behavior dependend, e.g., on the immediate caller. One of those methods is `Class.forName`. It will use the immediate caller's classloader to load a specified class. Another example is `Class.getDeclaredMethods`, which will skip a permission check if only the immediate caller is trusted.

A similar feature is the `MethodHandles` API. Even though implemented differently, it can also be used to obtain references to methods and fields of other classes during runtime. Consequently, this feature also poses a risk to the secure implementation of the Java platform.

Finally, we want to highlight deserialization as a risky feature. Serialization is the process of translating objects, including their field values, into a storable data format, e.g., a byte array. Deserialization is the reverse process of reconstructing such stored objects in the runtime. One of the risks involved in this is that attackers can craft arbitrary serialized objects. A deserialized object may be in a state that would be impossible to achieve without serialization. This is risky, because it may give attackers access to objects they would not normally have access to [17].

The analysis in Section 5 will refer to these features and provide more details on how they are relevant for Java security in practice.

3. EXPLOIT SAMPLE SET

We were interested in collecting a large and diverse set of exploits. To structure our efforts, we followed a multi-step process. First, we collected exploits from various online databases and exploit frameworks, including *Metasploit* (22 exploits)[3], *Exploit-DB* (2)[4], *Packet Storm* (5)[5], from the security research company *Security Explorations* (52)[6], and an online repository for Java exploits[7]. Thus we studied 87 exploits in total. The majority of 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. The associated CVE identifiers, where available, range from 2003 to 2013. We tagged all original exploits with a unique ID to allow for easy tracking throughout the analysis.

After our collection process, we tested all exploits in an isolated environment to verify that the exploits were actually effective. To do so, we created a common testing framework. For exploits that bypass the Java sandbox model, our framework sets the `SecurityManager` to the default `SecurityManager`, runs the exploit and checks whether the `SecurityManager` is set to *null*, afterwards. This allows for testing all such exploits in a uniform and fully-automated manner. For exploits that do not aim for disabling the `SecurityManager` we tested the effectiveness manually. 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 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 read because they contained large byte arrays with possible payloads. Some of them also contained GUI elements, unnecessary reflection constructs, bugs, etc. As a next step, 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,

[3] https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits, last accessed 2016-05-20
[4] https://www.exploit-db.com, last accessed 2016-05-20
[5] https://packetstormsecurity.com, last accessed 2016-05-20
[6] http://www.security-explorations.com/en/SE-2012-01-poc.html, last accessed 2016-05-20
[7] https://bitbucket.org/bhermann/java-exploit-library, last accessed 2016-08-03

```
1 // Method loads arbitrary classes
2 private Class getClass1(String s) {
3   JmxMBeanServer server=(JmxMBeanServer)
        JmxMBeanServer.
4     newMBeanServer("",null,null,true);
5   MBeanInstantiator i=server.
        getMBeanInstantiator();
6   return i.findClass(s,(ClassLoader)null);
7 }
```

Listing 1: Modified excerpt from exploit for CVE-2013-0431

```
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 }
```

Listing 2: Modified excerpt from exploit for CVE-2012-5088

we replaced the large byte arrays with the code they contained, creating the byte arrays on demand.[8] We did this reverse engineering to facilitate code reviews.

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, working exploits that we used as a basis for the analysis.

4. MODELING EXPLOIT BEHAVIOR

4.1 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 throughout this analysis.

Instead of providing an abstract, formal definition of the term, let us consider Listings 1 and 2. Both listings contain one method each, `getClass1` and `getClass2`, both of which are able to dynamically load a class. Since they make use of security vulnerabilities, an exploit can use them to load arbitrary classes, including restricted ones. As explained in Section 2, 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 `MethodHandle` instead. If some developer were to document the behavior of any of these methods, one intuitive way would be to add a code comment similar to the one in line 1 of Listing 1 and 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 of 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 typically requires more than one building block to adequately model its behavior on this level of abstraction. Further, note that the two code snippets shown rely on different vulnerabilities. Another purpose of identifying snippets with the same behavior is thus to see whether one could be replaced by another, e.g., when the vulnerability exploited in the first was fixed but not the one exploited by the second.

4.2 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 4.1) and abstract from implementation, i.e., specific bug details. Only with this layer of abstraction is it 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 to achieve their goal. The model must thus allow for documenting the use of intended functionality, such as the reflection API, or `MethodHandles`.

Guided by the above requirements, we developed a new meta model that we instantiated to document all exploits in the sample set. As can be seen in Figure 1, this model comprises the following seven entities:

[8] In some cases, however, this was not possible because the very nature of the exploit was to work with bytecodes that cannot be produced from source.

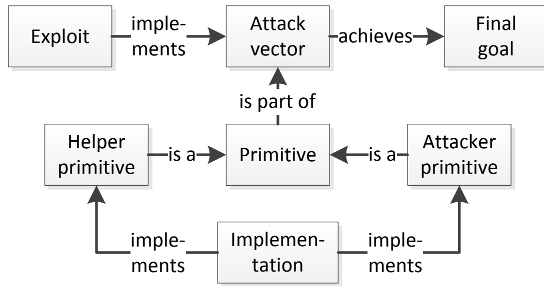

Figure 1: Meta model used to document exploits.

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.
Example: 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 ID, 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 behavior that was introduced through 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 behavior that was introduced as a feature of the target system, as opposed to attacker primitives, which were 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 is adding 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 implementatation instantiates only one primitive, but a primitive can have multiple implementations.
Example: The codes in Listing 1 and Listing 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 1, and another exploit uses the code of Listing 2 instead. If this is the only difference between those two exploits, we consider them to be different exploits that implement the same vector.

The meta model, as described above, is by no means specific to Java exploits. In fact, it is generalizable enough to be applied in entirely different attack domains.

4.3 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. 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. Since all exploits in this sample set are tested and unique, our analysis only considers reproducible attacks, we avoid duplicates, and we document only the minimal code that is actually needed to carry out an attack.

Documenting the sample set for our analysis requires us to instantiate the meta model. This means, we have to develop a set of final goals, primitives, attack vectors, etc. that closely resemble the behavior of the actual exploits. The meta model we developed just describes how to descriptively document exploit behavior, 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 [13] 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). The largest portion of exploits in the sample set aims for arbitrary code execution, often achieved through disabling the active security manager.
- 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 of describing exploit behavior is to document for each exploit the set of primitives it uses to achieve

its final goal. For this, we properly inspected all exploits in detail to understand which vulnerabilities and features they use to perform the attacks. This step required multiple iterations to ensure that all primitives we describe are on the same level of abstraction. It is obvious that there is a certain design space when it comes to choosing appropriate primitives to model exploit behavior. Those primitives are not given, and there is no ground truth. However, the specification of new 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 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 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. The Java platform is feature-rich, and implementing even just simple applications requires heavy usage of the JCL. However, not all parts of the class library used by exploits are of interest from a security point-of-view. To select relevant features, we consulted the list already presented in Section 2. Those are the features that either implement the Java security model, or pose a risk to the proper implementation of the model. We assume that those parts of the class library are more likely to point to design weaknesses.

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. Table 1 provides an overview of all primitives used for exploit documentation. 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 Section 5. We make the full documentation publicly available along with this paper[9].

5. 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, 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.

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

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 MethodHandle
H10	Private PrivilegedAction that provides access to arbitrary no-argument methods and sets them accessible
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

Table 1: Overview of the primitives used for exploit documentation. Helper primitives have an identifier starting with H, attacker primitives start with A.

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

Table 2: Overview of the weaknesses we identified and the number of minimal exploits that use them. One exploit can use more than one weakness.

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.

5.1 RQ1: What are the weaknesses attackers exploit to implement their attacks?

As can be seen in Table 2, we derived a set of nine weaknesses from the full documentation of the 61 minimal exploits. All weaknesses represent a specific kind of vulnerability or functionality used by at least 10% of all exploits. They are well-suited for providing an overview as they combine multiple related primitives and implementations. Note that some primitives are associated with more than one weakness, and that one exploit can make use of more than one weakness. In the following, we explain in detail how exploits make use of the nine weaknesses.

W1: Caller sensitivity.
Related primitives: H1, H3, H4, H5
Caller-sensitive methods vary their behavior depending on their immediate caller, e.g., skip permission checks if only the immediate caller is trusted. Such methods are abused by 22 minimal exploits for the following purposes.

- 22 exploits use methods, primarily `Class.forName`, to load arbitrary classes.
- 13 of the former 22 exploits also use caller-sensitive methods to get reflective access to members of classes, i.e., fields, methods, and constructors, they should not be allowed to access.

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 modelled 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.

```
1 Class A {
2     public Object invoke(Method m,Object[]
          args) {
3        return m.invoke(this,args);
4     }
5     // ...
6 }
```

Listing 3: Simplified example code to illustrate a confused-deputy vulnerability

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. An empirical evaluation of the implications of caller-sensitive methods on security and API usability is certainly required, however, we are unaware of published research in this area.

W2: Confused deputies.
Related primitives: A7, A17, A20, A21
Confused deputies that are part of the JCL can be used 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, caller-sensitive methods often behave differently depending only on the immediate caller, sometimes even skipping permission checks if the immediate caller is trusted. Thus, calling certain methods through a system class can be profitable to an attacker. 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. In this example, method `A.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 `A`. The second argument is an array of arguments. Just by reviewing this method, it seems impossible for any caller to use `A.invoke` to invoke a method outside of class `A`, as the first argument to `Method.invoke` is always `this`, pointing to an instance of `A`. 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 `A` 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.
- Four exploits abuse a defect in the implementation of MethodHandles. An example for this is presented in List-

ing 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 MethodHandles 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 a subset of all methods.

W3: Privileged code execution.

Related primitives: A18, A19, A25

We differentiate between privileged code execution and other confused deputies. The confused deputies we referred to in the previous paragraphs allowed untrusted code to route a call sequence through 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 dependend 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 [1]. In trusted method chaining, malicious code is able to setup a thread that will eventually execute code that was provided by the attacker, in such a way, that the malicious code itself is not on the call stack. 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 privileged 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 is indicating that explicit privilege elevation is easier to control than implicit privilege elevation, however, thorough empirical studies are needed to investigate this matter further.

W4: Loading of arbitrary classes.

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

Dynamic classloading is a central security-related feature of the Java platform. Classloaders 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 classloader to load the requested class. Since in this setting the immediate caller of forName is a trusted system class, and its defining classloader is the almighty bootstrap classloader, untrusted code can request the loading of arbitrary restricted classes. Listing 2 gives an example. We modeled the various confused-deputy defects as instances of attacker primitives, and the corresponding caller-sensitive methods as a helper primitive.

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 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 classloader. This special classloader will not define the class in a privileged context, however, the classloader itself allows for loading arbitrary classes. A custom class, that has been defined this way, can thus simply call Class.forName, which will use the caller's defining classloader, 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 classloading should be caller-sensitive. Removing them is possible, especially since there are non-caller-sensitive alternatives, e.g., Class.forName(String, boolean, ClassLoader). While their immediate removal would break backward compatibility, one should consider their deprecation. The remaining vulnerabilities that allow for arbitrary classloading are too diverse to be addressed by a single solution. To fix them, a major redesign of the classloading 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 fields 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 from 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 is instantiated with a platform-specific toolkit that in turn extends `sun.awt.SunToolkit`. Using `Class.getSuperClass` then provides access to `SunToolkit`.

Instead of using stack inspection, restricted classes are protected in a capability-based manner. 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 usage of restricted classes in the exploit sample set illustrates that this entire concept is very hard to implement securely. The fact that the number of restricted classes increased significantly over time is a dangerous trend. 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 maintainance 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. A potential alternative way of dealing with this problem could be the planned Java Module System [2], which may provide more effective ways of preventing untrusted code from using certain sensitive functionality. However, the Java Module System is still under development, and requires extensive security analyses before it can be considered a solution to the problem we describe.

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 a proper permission check. It may be possible to implement these changes even without breaking backward compatibility, however, this requires further investigation.
- Two exploits obtain unauthorized access to `MethodHandles` for arbitrary protected methods. This can be used to call internal methods of classloaders 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.

- Five of the nine exploits make use of serialization issues. Two of them use a deserialization sequence to instantiate a custom classloader, which can be used to define a class with higher privileges. Another exploit uses deserialization within a custom thread, to have a restricted class be loaded by the bootstrap classloader.

  Two exploits use serialization issues to bypass information hiding, but in different ways. One of the two exploits, involving CVE-2013-1489, prepares an instance of a system class in a way that would be impossible when running with limited privileges. Specifically, it manipulates the value of a certain private field of that system class, which holds a bytecode representation of a class that will later be defined by triggering a specific call sequence. This is profitable, because the system class will define this custom class in a namespace that provides access to restricted classes. An attacker would prepare the instance of that system class before the actual attack. When the exploit code is to be deployed, it only contains the serialized object. Deserialization of the manipulated instance is possible even when running with limited privileges.

  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 system 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 classloader with the application classloader, 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.

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, but also set them accessible. This is very valuable to an attacker, as it allows for using private members without requiring another vulnerability. Frequently used defects of this kind were found in `sun.awt.SunToolkit`.
- 13 exploits use confused deputies to invoke caller-sensitive methods, such as `getDeclaredFields` and `getDeclaredMethods` in `java.lang.Class`.
- Twelve exploits abuse restricted classes to access class members they should not be allowed to access, or set certain fields accessible.
- 13 exploits make use of other issues to access members.

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

W9: MethodHandles.

Related primitives: H2, H9, A6, A26

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

- Twelve vulnerabilities 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. Due to the capability-based design, malicious code does not have to bypass any security checks to get access to class members after the lookup object has been retrieved from the confused deputy.

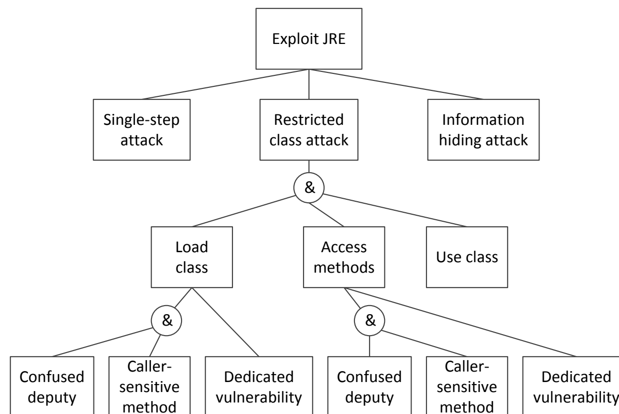

Figure 2: Shortened attack tree that illustrates the three categories of attack vectors we identified.

- 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 `MethodHandles`, as illustrated in Listing 2. In other cases, however, `MethodHandles` has been deliberately used as an alternative to the reflection API, because `MethodHandles` can be less strict when it comes 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 `MethodHandles` worked without any errors. While this flexibility of `MethodHandles` is advertised as a feature, it is also helpful to attackers.
- Three exploits use other vulnerabilities that will provide untrusted code access to MethodHandles that should be inaccessible.

5.2 RQ2: How do attackers combine the weaknesses to attack vectors?

The entire set of 61 minimal codes implements 33 different attack vectors. As explained in Section 4.2, a vector is a specific combination of primitives. Each primitive is implemented by specific code sequences, which we call *implementations*. 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 differerent implementations.

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 [16] in Figure 2, these categories are *single-step attacks*, *restricted-class attacks*, and *information-hiding attacks*. In the following, we will describe each category in detail.

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 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 `MethodHandles` for arbitrary protected methods. These single-step attacks are hard to mitigate systematically, as they exploit individual vulnerabilities of various types in different components.

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 2, 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 classloading API, or a confused-deputy vulnerability. A possible alternative way to get access to a restricted class is by retrieving it from a trusted class, either because it leaks an instance through a public interface, or because it holds an instance in a private/protected field, which can be accessed from untrusted code by means of another vulnerability. An example for such an attack is the exploit that involves CVE-2012-1726. It first uses a vulnerability to break information hiding, thus getting 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 `SunTookit` 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 can be seen, this vector does not involve the immediate use of a classloader or a confused deputy. However, we find that this is a rare case. Only four exploits that use restricted classes implement a vector that does not depend on classloading vulnerabilities or confused deputies.

The third category, information hiding 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 vulnerabilities to break information hiding. One of the two exploits combines two different vulnerabilities to achieve this. One is used to get access to declared fields of a class, and the other one to set them accessible. It uses this capability to set the private field `System.security` to `null`, thus disabling all security checks. The second exploit combines a vulnerability for loading arbitrary classes, with a vulnerability to call arbitrary public methods. To implement a full attack, it first creates an instance of `java.beans.Statement`, which targets `System.setSecurityManager` with a `null` argument. This alone does not violate the security model and cannot be used by untrusted code without causing an exception, because `Statement` will use the current `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 vulnerability that allows for loading arbitrary classes 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 the current `AccessControlContext`, which is used by `Statement` to perform the call. The exploit changes this field's value such that it holds a reference to another instance of `AccessControlContext` which has all permissions. After this modification, the `Statement` object can be successfully used to disable the security manager.

Discussion

Summarizing our findings, we can see that a large number of exploits benefit from design weaknesses. This includes the heavy usage of restricted classes, caller-sensitive methods in combination with confused deputies, as well as the inconsistencies between the reflection API and `MethodHandles`. There are also single-step attacks, which exploit individual security vulnerabilities introduced through implementation errors. However, a proper redesign of the aforementioned weakness areas, e.g., guarding sensitive functionality in restricted classes with permission checks, and removing caller sensitivity, could significantly improve the security of the Java platform in practice.

Some of the weaknesses presented in this paper may also be relevant to other platforms. The Microsoft .NET Common Language Runtime also uses stack-based access control to restrict access to sensitive resources. Consequently, all weaknesses related to this access control model are potentially relevant to the security of both platforms. This primarily includes issues with confused deputies (W2) and privileged code execution (W3).

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. show that serialization vulnerabilities allow an attacker to execute arbitrary code in many Android applications and services which could result in privilege escalation [12].

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 point to security weaknesses of the Java platform.

There are publications describing how Java exploits work at a very low and technical level. For instance, Fireye describes four Java vulnerabilities [4] and Oh studied a specific Java vulnerability that has been widely used for drive-by-download exploits [3]. Kaspersky Labs 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 [10].

Schlumberger et al. designed a tool to automatically detect malicious Java applets [15] using machine learning. Approaches of this kind require thorough feature selection, and our analysis of exploits could aid in the selection process.

Mastrangelo et al. studied the usage of `sun.misc.Unsafe` in practice [11]. One of their findings is that developers use it for performance optimization.

Several improvements have been proposed to overcome limitations of the classical approach to stack-based access control (SBAC). Abadi and Fournet propose History-Based

Access Control (HBAC), which extends SBAC by not only considering the methods currently on the call stack, but also all methods that completed execution before the permission check has been triggered [5]. Pistoia et al. propose 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 [14].

7. CONCLUSION

In this paper, we present 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 exploit frameworks, 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 dedicated 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.

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 platform, e.g., unauthorized use of restricted classes, arbitrary classloading, caller sensitivity, or `MethodHandles`. We explained in detail how attackers benefit from those weaknesses, and how they can be combined to full attack vectors. We found that there are three different catogories of attack vectors: (1) single-step attacks (46%), 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) information hiding attacks (3%), which use a combination of vulnerabilies to break information hiding in order to disable all security checks.

Finally, we proposed ideas for improving the security architecture to harden Java against future attacks similar to the ones observed so far. With this we hope to spawn further research in this area.

Acknowledgments

The authors wish to thank Marco Pistoia for his constructive feedback and Julian Dolby for providing us with the IBM JDKs. This work was supported by an Oracle Research Grant and by the DFG Priority Program 1496 Reliably Secure Software Systems through the project INTERFLOW.

8. REFERENCES

[1] Java trusted method chaining (cve-2010-0840/zdi-10-056). http://slightlyrandombrokenthoughts.blogspot.de/2010/04/java-trusted-method-chaining-cve-2010.html. [Online; accessed on 22-May-2016].

[2] The state of the module system. http://openjdk.java.net/projects/jigsaw/spec/sotms/. [Online; accessed on 22-May-2016].

[3] 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, 2012. [Online; accessed on 18-May-2016].

[4] 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, 2014. [Online; accessed on 18-May-2016].

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

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*, pages 241–250. ACM, May 2011.

[7] Security Exploration. [se-2012-01] broken security fix in ibm java 7/8. http://seclists.org/bugtraq/2016/Apr/19, 2016. [Online; accessed on 17-May-2016].

[8] Security Exploration. [se-2012-01] yet another broken security fix in ibm java 7/8. http://seclists.org/fulldisclosure/2016/Apr/43, 2016. [Online; accessed on 17-May-2016].

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

[10] Kaspersky Labs. Java under attack – the evolution of exploits in 2012-2013. https://securelist.com/analysis/publications/57888/kaspersky-lab-report-java-under-attack, 2013. [Online; accessed on 19-May-2016].

[11] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: the java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 695–710. ACM, 2015.

[12] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[13] CP Pfleeger and SL Pfleeger. Security in computing. 4th, 2007.

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

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

[16] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[17] John Viega, Gary McGraw, Tom Mutdosch, and Edward W. Felten. Statically scanning java code: Finding security vulnerabilities. *IEEE Software*, 17(5):68–74, 2000.