Give ’em less rope: Open Source Evidence of Java Sandbox Perversions

Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine
Carnegie Mellon University
{zfc,mmass}@cs.cmu.edu, tding@cmu.edu, {clegoues,sunshine}@cs.cmu.edu

ABSTRACT

The ubiquitously-installed Java Runtime Environment (JRE) executes untrusted code inside a sandbox to protect the host machine from potential malicious behavior. However, dozens of recent exploits have successfully escaped the sandbox, thereby enabling attackers to infect countless Java hosts. It is essential to distinguish patterns of malicious use from patterns of benign use to proactively prevent future exploits. We therefore performed an empirical study of benign open-source Java applications and compared their use of the sandbox to the usage present in recent exploits. We found that benign applications with secured sandboxes do not modify the security manager, the security policy enforcement mechanism, after it is first set and do not attempt to directly use privileged classes. Exploits routinely do both. We derive two rules from these results to prevent (1) security manager modifications and (2) privilege escalation. We evaluated their protection merits in a case study using runtime monitors to enforce the rules during the execution of exploits and benign applications. The rules stop all ten Metasploit Java 7 exploits without breaking backwards compatibility with benign applications. These practical rules should be enforced in the JRE to fortify the Java sandbox.

1. INTRODUCTION

The Java Runtime Environment (JRE) is widely installed on user endpoints, where it executes external code in the form of applets and Java Web Start (JWS) applications [1, 2]. These facts, combined with the hundreds of recently discovered vulnerabilities in Java, including zero-day vulnerabilities (e.g. CVE-2013-0422), have made Java a popular exploit vector (see Figure 1). Attackers typically lure users to websites containing hidden malicious applets. Once the user visits the website, the exploit triggers a series of events that ends with the delivery of malware, all while the user is left unaware. This kind of attack is commonly referred to as a “drive-by download.”

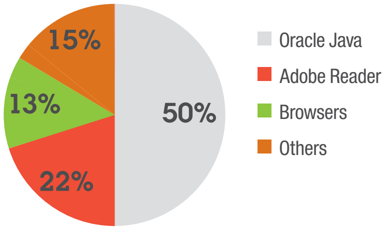

Figure 1: Pie chart showing the most targeted applications on enterprise workstations, according to to a Dec. 2013 survey of Trusteer customers [8]. Java represented half of all attack-attempts in their sample.

Java includes a mechanism to safely execute untrusted code and isolate components from one another in a sandbox, such that both the application and the host machine are protected from malicious behavior. However, the exploits cited above show that there is substantial room to improve the containment of code within the sandbox. Previous investigations of Java exploits have shown Java malware commonly alters the sandbox’s settings [3]. Typically, exploits disable the security manager, the component of the sandbox responsible for enforcing the security policy [4, 5, 6, 7]. We hypothesize that, when compared to the exploits, benign applications interact with the security manager differently. If true, this difference can be exploited to prevent future attacks.

To validate this insight, we conducted an empirical study of benign open source Java applications. Our empirical study was designed to answer the following research question: How do benign applications modify the security manager? To answer this question, we identified Java projects in the Qualitas Corpus [9] and the GitHub repository that make use of the security manager. We analyzed the resulting 46 projects using a custom FindBugs [10] plugin to isolate code involved in the initialization or modification of the security manager. We then manually characterized the security manager usage in each of the isolated code snippets. Finally, we used a Java Virtual Machine Tool Interface (JVMTI) agent to confirm that our characterizations were accurate at run time.

We discovered two types of security managers: *defenseless* security managers, which enforce a security policy that allows code inside the sandbox to modify sandbox settings, and *self-protecting* security managers, which disallow such behavior. Applications with defenseless security managers are inherently insecure. These applications sometimes modified or disabled the security manager during program execution. Some of these applications use the security manager to enforce policies unrelated to security. On the other hand, we found that applications with self-protecting security managers, a category which includes all applets and JWS applications, do not change sandbox settings during program execution.

Based on our analysis of benign and malicious applications, we propose two runtime rules to fortify the Java sandbox. The first rule mitigates privilege escalation by preventing restricted classes inside a sandbox from loading classes with fewer restrictions. The second prevents changes to the sandbox when a self-protecting security manager is initialized. We evaluated the protection merits of our rules by implementing each in runtime monitors used against ten applets in Metasploit 4.10.0[1] that successfully exploit unpatched versions of Java 7. The monitor for the privilege escalation rule detected and stopped four of the ten exploits. Using both monitors together detected and stopped all ten exploits. Neither monitor produces false-positives for a corpus of benign JWS applications.

Our results suggest that the rules are effective and should be enforced in the Java Virtual Machine (JVM) itself. This implementation strategy is motivated by two concerns: Existing mechanisms for monitoring the execution of Java applications are either (1) insufficient to securely enforce the rules (e.g. bytecode instrumentation) or, like JVMTI, (2) unacceptably degrade performance by disabling the just-in-time compiler (JIT). We are engaged in an on-going discussion on the security-dev mailing list for OpenJDK about implementing runtime enforcement of these rules in the JVM.

The contributions of this papers are as follows:

- An analysis of privilege escalation in the Java security model and recent Java exploits (Section 3).
- An empirical study of Java sandbox usage in benign, open-source applications (Sections 4 and 5).
- Two novel rules for distinguishing between benign and malicious Java programs (Section 6).
- A case study evaluation of the protection merits of our rules with a discussion of practical implementation considerations (Section 7).

2. BACKGROUND ON THE JAVA SANDBOX

In this section, we describe components of the Java sandbox that are relevant to understanding this work, how they compose to form the sandbox, and their functions. These points are summarized in Figure 2. The Java sandbox was designed to safely execute code from untrusted sources. Essentially, when a *class loader* loads a class from some location (e.g., network, filesystem, etc.) the class is assigned a *code source*. The assigned code source indicates the origin of the code and associates the class with a *protection domain*. Protection domains segment the application classes into groups, where each group is assigned a unique *permission set*. The permission sets contain permissions explicitly allowing actions with possible security implications, such as writing to the filesystem, accessing the network, using certain reflection features, etc. (see a more complete list at [11]). *Policies* written in the Java policy language [12] define permission sets and assign code sources to each set. By default, applications executed from the local file system are run without a sandbox, and all other applications are run inside a restrictive sandbox. This prevents applications from the network or other untrusted sources from executing malicious operations on the host system.

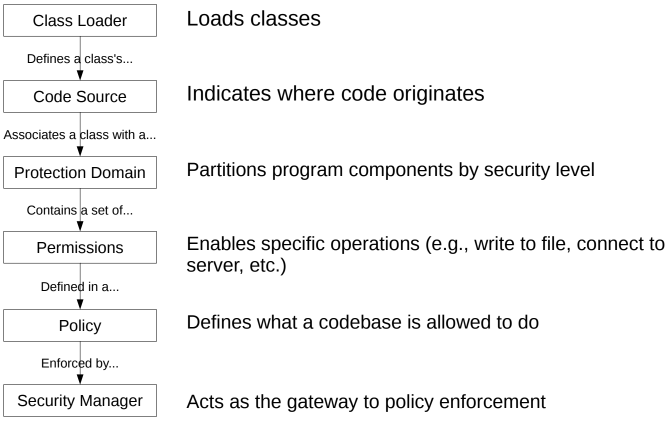
Class Loader
Loads classes
Defines a class's...
Code Source
Indicates where code originates
Associates a class with a...
Protection Domain
Partitions program components by security level
Contains a set of...
Permissions
Enables specific operations (e.g., write to file, connect to server, etc.)
Defined in a...
Policy
Defines what a codebase is allowed to do
Enforced by...
Security Manager
Acts as the gateway to policy enforcement

Figure 2: A summary of the components of the Java sandbox that are relevant to understanding this work.

Defined policies will not be enforced unless the sandbox is activated. The sandbox is activated by setting a security manager for the system. This security manager acts as the gateway between the sandbox and the rest of the application. Whenever a class attempts to execute a method with security implications inside a sandbox, the protected method queries the security manager to determine if the operation should be allowed. For example, if an application attempts to write to a file (e.g. `java.io.FileOutputStream`) inside a sandbox, the class that performs the write will check with the security manager to ensure that a write to that file is allowed. Missing checks are a common source of Java vulnerabilities because protected code must initiate the check.

To perform a permission check, the security manager walks the call stack to ensure each class in the current stack frame has the required permission. However, privileged code can stop stack walking before the entire frame has been walked by wrapping code inside a `doPrivileged` call. This allows privileged code sections to perform actions with security implications at the request of non-privileged code sections, once the request has been properly verified. If the permission check reaches a class in the stack frame that does not have the correct permissions, the security manager will throw a `SecurityException`. Stack-based access control is discussed in more detail in [13, 14, 15, 16, 17, 18, 19].

Java is flexible about when in an application's execution the sandbox is configured and enabled. The default case for web applets and applications that use Java Web Start is to set a *self-protecting* security manager before loading the application from the network. The security manager, and thus the sandbox, is self-protecting in the sense that it does not allow the application to change sandbox settings. A security manager can also be *defenseless*, which is the exact opposite of self-protecting. A defenseless manager does little to improve the security of a constrained application

[1] http://www.metasploit.com/

```
import java.lang.reflect.Method;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;

public class Payload
        implements PrivilegedExceptionAction {
    public Payload() {
        try {
            AccessController.doPrivileged(this);
        } catch(Exception exception) { }
    }

    public void run()\documentclass[10pt]{âĂć} {
        // Disable sandbox
        System.setSecurityManager(null);
    }

    public static void outSandbox() {
        // Do malicious operations
    }
}
```

Figure 3: A typical sandbox-disabling Java exploit payload from http://pastebin.com/QWU1rqjf.

or the host. However, we show in Section 5 that some benign applications have found interesting uses for defenseless managers.

Table 1 summarizes the set of permissions used to distinguish between self-protecting and defenseless security managers. We consider any security manager that enforces a policy that contains even one of the listed permissions to be defenseless. A subset of the permissions in this list were identified in [6].

3. EXPLOITING JAVA CODE

This section provides an analysis of privilege escalation in the Java security model and recent Java exploits. Between 2011 and 2013, drive-by downloads that used Java applets as the vector were widely reported. While the Java sandbox *should* prevent malicious applets from executing their payloads, exploits leveraged vulnerabilities in the Java Runtime Environment (JRE) to set the security manager to `null`. Setting the security manager to `null` disables the sandbox, allowing previously constrained classes to perform any operation that the JRE itself has the privileges to perform. Figure 3 shows a typical payload class whose privileges have been elevated by an exploit to allow it to disable the sandbox. This example payload uses `doPrivileged` to allow the unprivileged exploit class to execute the operations in the payload without causing a `SecurityException`.

Less than half of recent Java exploits use *type confusion* to bypass the sandbox. A type confusion vulnerability is exploited by breaking type safety, allowing the attacker to craft an object that can perform operations as if it is an instance of a class of a different type. For example, attackers craft objects that either (1) point to the `System` class or (2) act as if they have the same type as a privileged class loader (see CVE-2012-0507 [20]). In the first case the attack causes any operation performed on the masqueraded class to happen on the real `System` class, allowing the attacker to directly alter the field where the security manager is stored. In the second case the malicious class can load the exploit's payload with elevated privileges.

Another prominent class of Java exploits takes advantage of a *confused deputy* vulnerability [21], which is an example of privilege escalation. In the case of a confused deputy, the exploit often convinces a class with access to a vulnerable *privileged class* (i.e. a class with more privileges than the application's classes) to return a reference to it. The returned privileged class often contains a vulnerability such as a missing security check (e.g. where the class should consult with the security manager before performing some operation, but does not). In some cases, the privileged class may be directly accessible to all Java applications, but this is quite rare and typically the fault of a vulnerable third-party library. Providing all classes with direct access to a privileged class is a violation of the *access control* principle that is part of the Java development culture.[2] Once an exploit gains access to a vulnerable privileged class, that class can be tricked into executing code that disables the sandbox (see CVE-2012-4681 [22]).

For the most part, benign applications have no reason to directly access privileged classes. The majority of the JRE's privileged classes are internal implementations of features that applications can access via less-privileged code paths. For example, many reflection operations are implemented in the `sun.reflect` package, which has all permissions. However, Java applications are supposed to use classes in the `java.lang.reflect` package to use reflection and do not have direct access to the `sun` classes given default JRE configurations. Classes in the `java` package do not perform privileged operations themselves, but do have permission to access classes in the `sun` package.

A privileged class loader must be used to load a privileged class. Thus, a class typically does not have direct access to a class that has a vulnerability that can be exploited to bypass the sandbox unless the former had its privileges reduced at some point in the application's execution. This is implicit in the Java security model: If any class could load more privileged classes and directly cause the execution of privileged operations, the sandbox in its current form would serve little purpose. In sections 6 and 7 we discuss how we can leverage these distinctions to further fortify the sandbox.

Many of the recent type confusion and privilege escalation vulnerabilities would not have been introduced if the JRE were developed strictly following "The CERT Oracle Secure Coding Standard for Java" [23]. For example, Svoboda [5, 24] pointed out that CVE-2012-0507 and CVE-2012-4681 were caused by violating a total of six different secure coding rules and four guidelines.

In the typical case, following just one or two of the broken rules and guidelines would have prevented a serious exploit. For example, CVE-2012-4681 resulted from two rule violations in a privileged Abstract Window Toolkit (AWT) class in the `sun` package and two rule violations and an ignored guideline in a JavaBean class. The bean class was exploited to access the AWT class. The AWT class contained a method that reflectively fetched any field in any class, made the field public, and returned it. This is a violation of rule SEC05-J because reflection is being used to increase the accessibility of fields. It is also a violation of SEC00-J because the AWT class is privileged and leaks sensitive information (the fields) across trust boundaries. The AWT class should have followed all of the secure coding guidelines, but its violation of SEC00-J is especially problematic—the exploits use the leaked fields to disable the security manager.

[2] `https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm`

Table 1: List of sandbox-defeating permissions. A security manager that enforces a policy containing any of these permission is sufficient to result in a defenseless sandbox.

Permission	Risk
RuntimePermission("createClassLoader")	Load classes into any protection domain
RuntimePermission("accessClassInPackage.sun")	Access powerful restricted-access internal classes
RuntimePermission("setSecurityManager")	Change the application's current security manager
ReflectPermission("suppressAccessChecks")	Allow access to all class fields and methods as if they are public
FilePermission("<<ALL FILES>>", "write, execute")	Write to or execute any file
SecurityPermission("setPolicy")	Modify the application's permissions at will
SecurityPermission("setProperty.package.access")	Make privileged internal classes accessible

In the rest of this paper, we do not concern ourselves with the specifics of particular exploits. We will now explore how benign applications interact with the sandbox to define ways of delineating them from exploits.

4. INTRODUCTION TO THE SECURITY MANAGER STUDY

The nature of recent Java exploits caused us to ask the question: Do applications need access to any facility for disabling or weakening the sandbox? This study aims to answer this question and provide data in support of JVM enhancements to fortify the Java sandbox. We focus our efforts on the security manager, as it is the means by which applications interact with the sandbox.

Any JVM enhancements that are to stop even zero-day exploits while maintaining backwards compatibility with benign applications must be designed with an understanding of which operations both exploitive and benign applications perform on the security manager. Assuming there is a difference between the set of operations performed by exploits and those performed by benign applications, one can exclude the operations on which exploits depend, that are not of use to benign applications. This strategy would help ensure the sandbox continues to enforce its policy in a given execution without having to deal with the wide diversity in the manifestations of vulnerabilities within the JRE or the subtleties of their exploits. In this section we describe the methodology for and limitations of an empirical study that validated this strategy and answers our motivating question.

4.1 Prior work

Several recent studies have examined the use of security libraries and discovered rampant library misuse, which caused severe vulnerabilities. Georgiev et al. uncovered vulnerabilities in dozens of security critical applications caused by SSL library protocol violations [25]. These applications misconfigured high-level libraries such that the high-level libraries misused low-level SSL libraries which in turn failed silently. Somorovsky et al. demonstrate vulnerabilities in 11 security frameworks such that Security Assertion Markup Language (SAML) assertions are not checked properly when certain API mis-orderings are triggered [26]. Li et al. examined browser-based password managers and found that many of their features relied on an incorrect version of the same-origin policy, which could allow attackers to steal user credentials [27]. As far as we are aware no study has examined Java applications' use of the sandbox. Li Gong, the main designer of the Java security architecture, admitted in a ten year retrospective on Java Security that he didn't know how or how extensively the "fine grained access control mechanism" (i.e. the Java sandbox) is used [28]. We fill in that gap.

4.2 Methodology

As discussed in the previous sections, it is widely known within the Java security community that current exploits that operate on the security manager perform one operation: They disable it. To understand the operations benign applications perform on the manager, we undertook an empirical analysis consisting of static, dynamic, and manual inspections of the open source Java application landscape. More precisely, we answer the following research question: How do open source Java applications interact with the security manager? To answer this question, our empirical analysis aimed to validate four independent hypotheses. Each hypothesis is paired with a mitigation that can be implemented if the hypothesis is supported. The mitigations are given names that denote their relative strengths when compared to each other. For example, a "weak" mitigation stops a small number of in-scope exploits and is easily bypassed. An "ideal" mitigation stops all in-scope exploits and can never be bypassed. Our hypotheses and their accompanying mitigations follow:

Hypothesis 1: *Benign applications do not disable the security manager.* If this hypothesis holds, exploits can be differentiated from benign applications by any attempt to disable the current security manager. This **weak mitigation** would be easy to implement, but exploits that weaken the sandbox without disabling it would remain a threat. For example, attackers could potentially bypass the mitigation by modifying the enforced policy to allow the permissions they need or they could replace the current manager with one that never throws a `SecurityException`.

Hypothesis 2: *Benign applications do not weaken the security manager.* Validation of this hypothesis would enable mitigations that prevent attackers from weakening or disabling the sandbox. However, an implementation of this **moderate mitigation** would require differentiating between changes which weaken the sandbox and those that do not. Classifying changes in this manner autonomously is difficult because it requires context specific information that a general mitigation strategy may not have. For example, if a permission to write to a file is replaced by a permission to write to a different file, is the sandbox weakened, strengthened, or exactly as secure as it was before?

Hypothesis 3: *Benign applications do not change the sandbox if a self-protecting security manager has been set.* If supported, it is possible to implement a mitigation strategy that disallows any change to a security manager that is enforcing a strict policy (as defined in Section 2). To implement this **strong mitigation**, a runtime monitor must determine if a security manager is self-protecting at the time the manager is set. This can be easily achieved. While this mitigation has the same outcome as the moderate mitigation, it is significantly easier to implement soundly and it is therefore more likely to be effective in practice.

Hypothesis 4: *Benign applications do not change a set security manager.* If the study supports this hypothesis, any attempted change to an already established security manager can be considered malicious. The **ideal mitigation** could easily be implemented in the JVM.

Our empirical analysis used applications from the Qualitas Corpus (QC) [9] and GitHub to form a dataset of applications that use the security manager. To filter relevant applications out of the 112 applications in QC, we performed a simple `grep` of each application's source code to find instances of the keyword *SecurityManager*. When any instance of the keyword was found, we included the application in our dataset. This filtering reduced the set of applications to inspect from 112 to 29.

We performed a similar process on GitHub while searching only Java files. The keyword *System.setSecurityManager(* was added to remove false positives. To include applications that disable the manager we also searched for *System.setSecurityManager(null)*. We picked the top seven applications from the results for each keyword, removed false positives, and ended up with an additional 17 applications that were not already in QC. We only looked at the latest commit.

With the dataset in hand, we created static and dynamic analysis tools to assist in the manual inspection of each application. Our static analysis tool is a FindBugs [10] plugin that uses a dataflow analysis to determine where `System.setSecurityManager()` is called, as well as the lines of code where its arguments were initialized. We created a dynamic analysis tool using the Java Virtual Machine Tool Interface (JVMTI) [29]. JVMTI is designed to allow tools to inspect the current state of Java applications and control their execution; it is commonly used to create Java debugging and profiling tools. Our dynamic analysis tool set a modification watch on the `security` field of Java's `System` class. This field holds the current security manager object for the application. The watch prints out the class name, source file name, and line of code where any change to the field took place. A special notice is printed when the field is set to `null`.

We split the dataset between two reviewers. The reviewers both analyzed applications using the following steps:

1. Run grep on all Java source files in the application to output every line containing the keyword *SecurityManager* and the 5 lines before and after it.

2. Reject any application where it is clear from the grep output that the keyword is used in ways that are unrelated to the security manager class.

3. Run the static analysis on retained applications.

4. Manually inspect code specified in step 3's findings, starting with the line where the manager was set and tracing the code back to locations where potential security managers were initialized.

5. Manually inspect all of the lines returned in step 1, looking for how the application interacts with the sandbox.

6. Execute the application with the dynamic analysis using parameters and actions steps 4 and 5 show affect the security manager to verify conclusions from previous steps.

7. Summarize the operations performed on the security manager with an emphasis on points that support or reject each hypothesis.

We undertook a pilot study where each reviewer independently inspected the same six applications and compared their results. This ensured reviewers understood the analysis steps and produced consistent results.

4.3 The Security Manager Dataset

The Qualitas Corpus is a curated collection of open source Java applications for use in reproducible software studies. We investigated sandbox interactions in 29 applications from QC version 20130901.

While QC provides a strong starting point for the construction of a dataset for this study, their inclusion criteria[3] leads to the inclusion of large, popular applications and frameworks. We diversified our dataset by turning to GitHub. Table 2 lists all studied applications. Version numbers and Git commit hashes are available in an online supplement.[4]

4.4 Limitations

4.4.1 Internal Validity

Our results are dependent on accurately studying the source code of applications and their comments. In most cases, security manager interactions are easily understood, but there are a few particularly complex interactions that may be misdiagnosed. Furthermore, we did not review all application code, thus we may have taken a comment or some source code out of context in larger applications. Finally, using two different reviewers may lead to variations in the interpretations of some of the data.

We mitigated these threats by using a checklist, FindBugs plugin, and JVMTI agent to provide reviewers consistent processes for reviewing code and validating their results. Furthermore, we inspected entire source files that contained security manager operations. We tested our tools and processes in a pilot study to find and mitigate sources of inconsistencies.

4.4.2 External Validity

The study only includes open source applications. It is possible that closed source applications interact with the security manager in ways that we did not see in the open source community. However, we inspected a few small applications with our aerospace collaborators. We did not find any code that suggested this is the case.

4.4.3 Reliability

While the majority of the study is easily replicable, GitHub search results are constantly changing. Using GitHub to

[3] http://qualitascorpus.com/docs/criteria.html
[4] http://goo.gl/dtcqTM

Table 2: **Table of applications included in the security manager study.**

Application Name	Description	Repo
(Apache) Ant	Java Project Builder	QC
(Apache) Batik	SVG Image Toolkit	QC
C-JDBC	DB Cluster Middleware	QC
Compiere	Business Tools	QC
(Apache) Derby	Relational Database	QC
DrJava	IDE	QC
Eclipse	IDE	QC
FreeMind	Mind-Mapping Tool	QC
Galleon	Media Server	QC
(Apache) Hadoop	Distrib. Comp. Frwk.	QC
Hibernate	Obj.-Rel. Mapper	QC
HyperSQL	SQL DB	QC
JBoss	Application Middleware	QC
JRuby	Ruby Interpreter	QC
(Apache) Lucene	Search Software	QC
(Apache) MyFaces	Server Software	QC
NekoHTML	HTML Parser	QC
Netbeans	IDE	QC
OpenJMS	Messaging Service	QC
Quartz	Job Scheduler	QC
QuickServer	TCP Server Frwk.	QC
Spring Framework	Web Dev. Library	QC
(Apache) Struts	Web Dev. Library	QC
(Apache) Tapestry	Web Dev. Library	QC
(Apache) Tomcat	Web Server	QC
Vuze	File Sharing App.	QC
Weka	Machine Learning Algs.	QC
(Apache) Xalan	XML Trans. Library	QC
(Apache) Xerces	XML Parsing Library	QC
AspectJ	Java Extension	Github
DemoPermissions	Spring Extension	Github
driveddoc	Application Connector	Github
FileManager-FtpHttpServer	FTP Server	Github
Gjman	Development Toolkit	Github
IntelliJ IDEA	IDE	Github
Jmin	Lightweight JDK	Github
MCVersion-Control	Minecraft Utility	Github
NGOMS	Business Tools	Github
oxygen-libcore	Android Dev. Lib.	Github
refact4j	Meta-model Prog. Frwk.	Github
Security-Manager	Alt. Security Manager	Github
Spring-Modules	Spring Extension	Github
System Rules	JUnit Extension	Github
TimeLag	Sound Application	Github
TracEE	JavaEE Support Tool	Github
Visor	Closure Library	Github

Table 3: Classification of application interactions with the security manager.

Type of Interaction	QC	GitHub	Total
1. Sets manager, nothing else	6	1	7
2. Changes set security manager	5	3	8
3. Interactions in production code	10	3	13
4. Interactions only in unit tests	3	5	8
5. No interactions (false positive)	5	5	10

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new
              RMISecurityManager());
```

Figure 4: Weka code snippet that initializes a security manager if one does not already exist. This is the only reference to the security manager in the Weka code base.

generate a new dataset using our method would likely generate a different dataset. Furthermore, over the course of our security manager study, two applications either became private repositories or were removed from GitHub (FileManagerFtpHttpServer and Visor).

5. SECURITY MANAGER STUDY RESULTS

We divided the security manager dataset into categories based on the operations each application performed on the security manager. The categories are summarized as follows: (1) applications that set a security manager that does not get changed later in the application's execution, (2) applications that change a set manager at some point in the program's execution, (3) applications that interact with a security manager in production code if one is set, (4) applications that only interact with the manager in unit tests, and (5) false positives that do not actually interact with the manager. Table 3 shows how each application in our dataset was categorized. The number for each category type in the table corresponds to the numbers in the previous list used throughout the rest of this section.

Type 1 applications set a security manager that is not changed during any execution of the application after it is set. In other words, for each possible execution path, there is at most one place the application sets a security manager. For example, Weka contains several main methods, most of which never set a security manager. However, the main method in `RemoteEngine.java` sets a security manager, as shown in Figure 4, unless the environment set one already (e.g. the user set one on the command line, or Weka is running as an applet or web start application). One Type 1 application, JTimeLag, did not actually set a security manager, but did set the security manager to null as discussed in Section 5.2.2.

Type 2 applications are of particular interest in validating our hypotheses because they change a set security manager at some point in the application's execution. In other words, the weak, moderate, and ideal mitigations break these applications. Due to their effect on our hypotheses, applications of this type are discussed in Sections 5.2 and 5.3.

Type 3 applications contain code that enables them to interact with a security manager if one is set, but never actually set a security manager themselves. These applications contain code that either (A) performs permission checks if the application is sandboxed or (B) uses privileged actions [30] to ensure the application works if constrained. Similarly, Type 4 applications contain unit test code that ensures the application works correctly if sandboxed or that set a manager themselves. These applications are not useful for validating our hypotheses because their interactions with the manager are not in production code.

Type 5 primarily includes applications that have a class whose name contains the word “SecurityManager” but do not extend the `SecurityManager` class. These custom classes cannot be used to enforce a JRE-wide security policy, thus applications of this type are false positives.

The remainder of this section provides details about Type 1 and Type 2 applications, with few details about the remaining types because they did not have a significant bearing on the outcomes of this study.

5.1 Evaluation of the Hypotheses

We only require one counterexample to falsify a hypothesis from Section 4. This section summarizes how our hypotheses held up against the results of this study.

Hypothesis 1: *Benign applications do not disable the security manager.* The investigation determined that some benign applications do disable the security manager, which turns off the sandbox. The applications that explicitly disabled the manager typically were not using the sandbox for security purposes; these cases are further explained in Section 5.2. However, some of these applications turned off the sandbox temporarily to update the imposed security policy.

Hypothesis 2: *Benign applications do not weaken the security manager.* This hypothesis was not definitively falsified if turning off the security manager is excluded from weakening. However, multiple applications provided methods for the user to dynamically change the security policy or the manager. These methods did not restrict their callers from weakening the manager during execution.

Hypothesis 3: *Benign applications do not change the security manager if a self-protecting security manager has been set.* This hypothesis was supported by both datasets. No applications in our set change the sandbox settings after a self-protecting security manager is set.

Hypothesis 4: *Benign applications do not change a set security manager.* This hypothesis was shown to be false: multiple applications changed the security manager, both for security and non-security reasons.

In short, the strong mitigation is the only proposed mitigation that can be implemented without breaking benign applications.

5.2 Non-security uses of the Sandbox

This section describes applications that violated hypotheses 1, 2 and 4 while using the sandbox for reasons that do not satisfy security requirements. We found several applications using the sandbox in ways unrelated to system security. Most of these applications used the sandbox to enforce architectural constraints when interacting with other applications or forcibly disabled the sandbox to reduce development complexity.

5.2.1 *Enforcing Architectural Constraints*

Java applications commonly call `System.exit()` when a non-recoverable error condition occurs. This error handling strategy causes problems when an application uses another application that implements this strategy as a library. When the library application executes `System.exit()`, the calling application is closed as well because both applications are running in the same JVM. In many cases, this is not the intended outcome.

To prevent this outcome without modifying the library application, the calling application needs to enforce the architectural constraint that libraries can not terminate the JVM. In practice, applications enforce this architectural constraint by setting a security manager that prevents `System.exit()` calls. If a manager has already been set, applications tend to save a copy of the current manager before replacing it with one that prevents termination of the JVM, but defer to the saved version for all security decisions that do not have to do with enforcing this particular constraint. The original security manager is often restored when the library application is finished executing.

```java
System.setSecurityManager(new
    AntSecurityManager(originalSM, Thread.
    currentThread()));
...

getCurrentProject().executeTargets(targets);
    \\Note: Ant is executed on this line
...

finally {
...

    if (System.getSecurityManager()
        instanceof AntSecurityManager) {
        System.setSecurityManager(originalSM)
            ;
    }
```

Figure 5: Snippet of Eclipse code that uses a security manager to prevent Ant from terminating the JVM when Ant encounters an unrecoverable error.

This case appears in Eclipse, which uses Ant as a library. When an unrecoverable error condition occurs, Ant kills the JVM to terminate execution of the build script currently running. However, Eclipse should continue executing and report an error to the user when Ant terminates with an error code. Figure 5 shows how Eclipse sets a security manager to enforce this constraint right before Ant is executed. After Ant closes and any error conditions are handled, the original manager is restored.

GJMan also enforces this architectural constraint, as shown in Figure 6. A code comment (not shown) references a blog post that we believe is the origin of this solution.[5] The code contains an `apply` method that creates and sets a security manager to prevent termination of the JVM and an `unapply` method to disable the sandbox. GJMan is a library and does not use these methods itself, but applications that use it could.

In total, we found 3 applications that use a variation of this technique: Eclipse, GJMan, and AspectJ. While this technique does enforce the desired constraint, and appears to be the best solution available in Java at the moment, it may cause problems when applications are also using the sandbox for security purposes. The technique requires the application to dynamically change the security manager. This operation requires that the manager itself be defenseless or that the application is very carefully written to prevent malicious code from changing the manager or the policy it enforces. Defenseless security managers are not capable of reliably enforcing a serious security policy.

5.2.2 *Reducing Web Application Development Complexity*

[5] http://www.jroller.com/ethdsy/entry/disabling_system_exit

```
public static void apply() {
    final SecurityManager securityManager =
        new SecurityManager() {
        public void checkPermission(
            Permission permission) {
            if(permission.getName().
                startsWith(”exitVM”)) {
                throw new Exception();
            }
        }
    };
   System.setSecurityManager(securityManager)
       ;
}
public static void unapply() {
    System.setSecurityManager(null);
}
```

Figure 6: Methods in GJMan that enable and disable the sandbox to prevent termination of the JVM when select code is running.

We found applications that were complicated by the Java security policies for web applications (applets and applications launched via Java Web Start) [31]. In the latest versions of the JRE (1.7.72 and 1.8.25), users are prompted to allow the execution of Java web applications originating from a source other than the host machine. By default, Java executes such an application inside a restrictive sandbox that severely limits the operations the application can perform, excluding operations such as accessing local files, retrieving resources from any third party server, or changing the security manager.

For some applications, required behaviors may be prevented by the restrictive sandbox. To avoid executing the applet in a restrictive sandbox, a developer must first get the application digitally signed by a recognized certificate authority. Once the application has been properly signed, the developer may specify that the application should run outside of the sandbox. Before running this kind of application, the user must accept two prompts: one to run the application, and a second to run outside the sandbox.

If an application is executed inside a restrictive sandbox and uses permissions prevented by the restrictive sandbox, the application developer has two options: adjust the application to not require the restricted permissions or have the application terminate with a `SecurityException`. The first option would require the application to recognize the sandbox and then avoid using restricted permissions. Thus, the application would only be able to execute a subset of the full application. This requires more development effort because the application is essentially implemented twice, once as the restricted version and again as the full version. The second option does not allow the application to execute inside a restrictive sandbox: The application must run outside the sandbox. We found that applications using this method attempted to set the security manager to `null` at the beginning of the application, causing the sandbox to catch the security violation and terminate the application.

We found two applications that use the second option: Eclipse and Timelag. Figure 7 shows a snippet from Eclipse's WebStartMain.java file that performs this operation. The comment shows that Eclipse attempts to disable the sandbox to avoid the permission issues caused by the default sandbox for web start. Timelag performs the same operation in the file JTimelag.java but does not contain any comments, thus we can only infer the motivation behind turning off the sandbox.

```
/**
* The launcher to start eclipse using
    webstart. To use this launcher, the
    client
* must accept to give all security
    permissions.
...

public static void main(String[] args) {
System.setSecurityManager(null); //TODO Hack
        so that when the classloader loading the fwk is
        created we don't have funny permissions. This
        should be revisited.
```

Figure 7: A snippet from Eclipse that disables the sandbox when Java Web Start is used to run the IDE.

```
public void enforceSecurity(boolean enforce){
SecurityManager sm = System.
    getSecurityManager();

if (sm != null && sm !=
    lastSecurityManagerInstalled){
    ...

    throw new SecurityException
        (Messages.getString(
            EXCEPTION_ALIEN_SECURITY_MANAGER)
            );
}
if (enforce) {
    ...

    installSecurityManager();
} else {
    if (sm != null) {
        System.setSecurityManager(null);
        lastSecurityManagerInstalled = null;
        ...
```

Figure 8: Security manager interactions in Batik.

5.3 Using the Security Manager for Security Purposes

This section describes applications that violated hypotheses 1, 2, and 4 while using the sandbox in attempts to improve the security posture of the application. Batik, Eclipse, and Spring-modules provide methods that allow the user to set and change an existing manager, and Ant, Freemind, and Netbeans explicitly set then change the manager.

Figure 8 shows an interesting case from Batik copied from `ApplicationSecurityEnforcer.java`. This method allows users to optionally constrain the execution of an an application that uses the Batik SVG Toolkit. It takes a parameter that acts as a switch to turn the sandbox on or off. The download page on the Batik website shows several examples of how to use the library. Two set a security manager at start up: the squiggle browser demo and the rasterizer demo. While the squiggle browser demo sets a manager and never changes it, the rasterizer demo calls `enforceSecurity` with a true argument the first time and a false argument the second time, which enables then disables the sandbox.

```
<permissions>
  <grant class="java.security.AllPermission"/>
  <revoke class="java.util.PropertyPermission"/>
</permissions>
```

Figure 9: Example Ant build script element to grant all but one permission. This specific permission set leads to a defenseless security manager.

```
public synchronized void setSecurityManager()
      throws BuildException {
  origSm = System.getSecurityManager();
  init();
  System.setSecurityManager(new MySM());
  active = true;
}

/**
  * Initializes the list of granted
      permissions, checks the list of revoked
      permissions.
  */    private void init() throws
      BuildException {
  ...

public synchronized void restoreSecurityManager() {
  active = false;
  System.setSecurityManager(origSm);
}
```

Figure 10: Code snippet showing Ant's custom security manager architecture, used to allow builds to run Java code under a specific set of permissions unique to the instantiated code.

While this was an interesting occurrence, there seems to be no valid reason to disable the sandbox in this case other than to show off the capability to do so.

Ant, Freemind, and Netbeans explicitly set and then change the manager during runtime. Ant allows the users to create build scripts that execute Java classes and JAR's during a build under a user specified permissions set. The permissions set is defined in the `permissions` element of an Ant build file. Figure 9 shows the first example from the Ant Permissions website.[6] The contents of the `grant` element provide the application all permissions, but the contents of the `revoke` element restrict the application from using all property permissions. Due the use a defenseless security manager, malicious code can easily bypass the restrictions in this example by executing the line `System.setSecurityManager(null)`. Once the sandbox is disabled, the malicious code can perform all actions including those requiring `PropertyPermissions`. The Ant Permissions website does not mention that an implementation of this example is vulnerable. The site also does not mention that allowing certain permissions will potentially enable the restricted application to bypass any user specified restrictions.

Ant uses the code section from `Permissions.java` shown in Figure 10 to provide the user the ability to grant and revoke permissions. Ant calls the `setSecurityManager` method of the `Permissions` class when it is about to execute the restricted application. This method starts by saving the current security manager to a temporary variable. Then the `init` method creates the set of granted permissions. The `setSecurityManager` method then initializes a custom security manager, `MySM`, which enforces the permissions specified by the user. When the security manager is initially set, it allows all permission checks to succeed, making it a defenseless security manager. Once the `setSecurityManager` method changes `active` to true, `MySM` enforces the specified permissions. When Ant has finished executing the application, it calls the method `restoreSecurityManager` to restore the original security manager.

```
public void checkPermission(Permission pPerm,
      Object pContext) {
  if(mFinalSecurityManager == null) return;
  mFinalSecurityManager.checkPermission(pPerm,
        pContext);
}
```

Figure 11: Code snippet showing how Freemind's custom security manager selectively enforces permissions.

With this implementation, Ant catches applications that perform actions restricted by the user while typically protecting sandbox settings. Ant is also able to revert to the original security manager after executing the application. However, we are not sure that this implementation is free of vulnerabilities.

Netbeans similarly sets a security manager around a separate application. Both of these cases require a defenseless security manager, otherwise the application would not be able to change the current security manager. However, this technique may cause problems if Ant or Netbeans are used in security critical settings, which require a self-protecting security manager. A better implementation would use a custom class loader to load the untrusted classes into a constrained protection domain. This approach would align with the intended usage of the sandbox. Additionally, it would be more clearly correct and trustworthy while allowing Ant and Netbeans to run inside of a self-protecting sandbox.

Freemind 0.9.0 tried to solve a similar problem but ended up illustrating the dangers of a defenseless manager. Freemind is a mind mapping tool that allows users to execute Groovy scripts on an opened map. The scripts are written by the creator of the mind map. Groovy is a scripting language that is built on top of the JRE: A Java application that executes a script typically allows the script to execute in the same JVM as the application itself. As a result, a mind map could potentially be crafted to exploit a user that opens the map and runs its scripts.

Freemind attempted to implement an architecture that would allow the sandbox to enforce a stricter policy on the Groovy scripts than on the rest of Freemind. Their design centers around the use of a custom security manager that is set as the system manager in the usual manner. This custom manager contains a field, `mFinalSecurityManager`, that specifies the proxy manager to be used during the execution of scripts. In this design, all checks to the security manager are ultimately deferred to the proxy manager set in this field, as shown in the example permission check in Figure 11. When this field is set to `null`, the sandbox is effectively disabled even though the system's manager is still set to the custom manager.

Figure 12 shows how Freemind sets the proxy security manager field in the file `FreemindSecurityManager.java`.

[6] https://ant.apache.org/manual/Types/permissions.html

```
/**
 * By default, everything is allowed.
 * But you can install a different security controller once,
 * until you install it again. Thus, the code executed in
 * between is securely controlled by that different security manager.
 * Moreover, only by double registering the manager is removed. So, no
 * malicious code can remove the active security manager.
 *
 * @author foltin
 *
 */
public void setFinalSecurityManager(SecurityManager pFinalSecurityManager) {
    if(pFinalSecurityManager == mFinalSecurityManager){
  mFinalSecurityManager = null;
      return;
   }
   if(mFinalSecurityManager != null) {
   throw new SecurityException("There is a SecurityManager installed already.");
   }
   mFinalSecurityManager = pFinalSecurityManager;
 }
```

Figure 12: Initialization of the field in Freemind's custom security manager that stores the proxy security manager.

```
        public void checkSecurityAccess(String pTarget) {
 }
```

Figure 13: Freemind's custom security manager overrides permissions checks they are not concerned with, then fails to implement them. This effectively grants all permissions associated with the check.

Once a manager is set, if `setFinalSecurityManager` is called again with a different security manager, a `SecurityException` is thrown, but calling the method with a reference to the set manager disables the sandbox. The comment implies this specific sequence of operations was implemented to prevent malicious applications from changing the settings of the sandbox.

The Freemind code responsible for initiating the execution of the Groovy scripts sets a proxy security manager that does not allow unsigned scripts to create network sockets, access the file-system, or execute programs on the machine. The manager explicitly allows all other permissions. Figure 13 shows a sample permission check from the proxy manager in `ScriptingSecurityManager.java`. `checkSecurityAccess` typically checks if the sandbox may be changed but due to the its empty implementation the check will always succeed. A malicious script can call `System.setSecurityManger(null)` to turn off the sandbox at any point.

We demonstrated that the custom security manager is easily removed using reflection to show that the problem is more complex than simply fixing this permission check. Figure 14 shows an exploit to turn off the manager written as a Groovy script in a mind map. The script gets a reference to the system's manager and its class. The class has the same type as the custom security manager, thus the exploit gets a reference to the proxy manager field. The field is made public to allow the exploit to reflectively `null` it, disabling the sandbox. Finally, a file is created to demonstrate that a "forbidden" operation succeeds.

```
def sm = System.getSecurityManager()
def sm_class = sm.getClass()
def final_sm = sm_class.getDeclaredField("mFinalSecurityManager")
final_sm.setAccessible(true)
final_sm.set(sm, null)
new File("hacked.txt").withWriter { out -> out.writeLine("HACKED!") }
```

Figure 14: Example exploit that breaks out of the scripting sandbox in Freemind to execute arbitrary code.

We sent a notice to the Freemind developers in August of 2014 to provide them with our example exploit and to offer our advice in achieving their desired outcome.

5.4 Security Manager Interactions are Potentially Dangerous

We found an interesting reference to the security manager in IntelliJ IDEA Community Edition. The IntelliJ editor contains static analysis checks, called inspections,[7] to warn users of potentially problematic sections of code. There are two security inspections that directly pertain to interactions with the sandbox: one that highlights calls to `System.setSecurityManager()` and another that highlights the definition of custom security manager classes. The descriptions for these inspections respectively contain warnings that imply interactions with the manager can create security issues:

- "While often benign, any call to System.setSecurityManager() should be closely examined in any security audit.“
- "While not necessarily representing a security hole, such classes should be thoroughly and professionally inspected for possible security issues.“

IntelliJ's concerns with these types of security manager interactions are warranted given our findings in the previous section.

6. RULES FOR FORTIFYING THE SANDBOX

Given the results of our investigation in Section 4 and the discussion in Section 3, we can fortify the sandbox for applications that set a *self-protecting* security manager. In this section, we define two rules to stop Java exploits from disabling the manager. These rules are backwards-compatible with benign applications: the Privilege Escalation rule and the Security Manager rule.

6.1 Privilege Escalation Rule

The *Privilege Escalation rule* ensures that, if a self-protecting security manager is set for the application, a class may not directly load a more privileged class. This rule is violated when the protection domain of a loaded class implies a permission that is not implied in the protection domain that

[7] http://www.jetbrains.com/idea/documentation/inspections.jsp

loaded it. About half of recent exploits break this rule to elevate the privileges of their payload class.

If all classes in the Java Virtual Machine (JVM) instance were loaded at the start of an application, this rule would never be broken. However, the JVM loads certain classes on demand, and some of the JVM classes have the full privileges. The rule makes exceptions for classes in packages that are listed in the `package.access` property of `java.security.Security` as these classes are intended to be loaded when accessed by a trusted proxy class.

6.2 Security Manager Rule

The *Security Manager rule* states that the manager cannot be changed if a *self-protecting* security manager has been set by the application. This rule is violated when code causes a change in the sandbox's configuration, the goal of many exploits. This rule is an implementation of the strong mitigation.

7. VALIDATING THE RULES

In Section 4, we discussed four hypotheses about security manager usage in benign applications, each of which, if validated, leads to a distinct mitigation. In Section 5, we gave empirical evidence in support of Hypothesis 3 and rejected all of the others. Along the way, we learned practical lessons about how applications use the Java sandbox that are useful to exploit mitigation implementers.

In this section, we evaluate the protection merits and backwards compatibility of the rules presented in Section 6 through an implementation of runtime monitors that enforce them. This evaluation was done in collaboration with a large aerospace company.

Section 7.1 discusses how we implemented our runtime monitors using JVMTI. Section 7.2 explains the methodology behind and results of an experiment we conducted to determine how effective the rules are at stopping existing exploits without breaking benign applications. Finally, Section 7.3 covers prior work related to Java exploit mitigations.

7.1 Implementation Using JVMTI

JVMTI is a native interface used to create analysis tools such as profilers, debuggers, monitors, and thread analyzers. Tools that use JVMTI are called agents, and are attached to a running Java application at a configuration-specific point in the application's lifecycle. The interface allows an agent to set capabilities, enabling the tool to intercept events such as class or thread creation, field access or modification, breakpoints, etc.

Our agent must intercept three events to enforce the Privilege Escalation and Security Manager rules: `ClassPrepare`, `FieldAccess`, and `FieldModification`. Enforcement of these rules is discussed in detail in subsections 7.1.1 and 7.1.2.

The field events require JVMTI to turn off the JIT, which slows down program execution enough that our monitors are not suitable for adoption on their own. JVMTI implementations can avoid this limitation, but avoidance would likely increase implementation complexity beyond what is reasonable for a diagnostic interface. We are currently in communication with the OpenJDK developers on their security-dev mailing list regarding enforcement of our rules in the JVM itself to avoid overhead issues.[8]

7.1.1 *Enforcing the Privilege Escalation Rule*

The Privilege Escalation rule is enforced by ensuring that classes can only load or cause the loading of more privileged classes in restricted-access packages after a self-protecting security manager has been set. *Restricted-access packages* are packages that are public but not intended to be directly used by typical Java applications; they are meant for internal JRE use only. These packages are listed in the `package.access` property in the `java.security.Security` class. There are two ways to unsafely and directly access packages listed in this property: (1) exploit a vulnerability in a class that can access them or (2) allow access via the `accessClassInPackage` permission.

Applications use JRE classes which call restricted access package classes. Thus, we must allow JRE to load restricted-access packages at runtime. For example, many of the classes in the `java.lang.reflect` package are backed by classes in the `sun` package, which is a restricted-access package containing the internal implementations for many Java features. However, enforcing the Privilege Escalation rule prevents exploits from elevating the privileges of their payloads because the payloads can not be in restricted-access packages with default JRE configurations.

To enforce the Privilege Escalation rule, our agent registers for the `ClassPrepare` event, which allows the agent to inspect a class after it is fully loaded but just before any of its code is executed. Assuming the loaded class is not in a restricted-access package, the agent inspects the stack frame to determine which class caused the new class to be loaded. The agent must get the protection domains for both classes. This can not be done from the agent using the JNI because the required Java calls[9] will be performed with the same permissions as the executing Java application. Most applications where this operation is relevant (i.e. those that have a self-protecting manager) do not have the necessary permission[10] to get a protection domain because the permission would allow a malicious class to probe the policy to determine which, if any, malicious operations it can perform. Because JVMTI agents are loaded into the JRE as a shared library, we instead load libjvm.so (jvm.dll on Microsoft Windows) to call JVM functions without security checks. Our agent leverages this ability to call the `GetProtectionDomain` JVM function to get the protection domains.

With both protection domains, the current agent implementation simply checks to see whether the loaded class's protection domain has all permissions while the class that caused the loading does not. If the latter is true, the Privilege Escalation rule has been violated. This specific check was used because it is fast, simple, and all privileged classes allow all permissions under known circumstances. It would be easy to update this check to instead ensure that every permission in the loaded class's protection domain is also implied by the other protection domain to handle other cases.

7.1.2 *Enforcing the SecurityManager Rule*

[8] REVIEWERS: This discussion will be referenced in the final version of the paper. At this time, the OpenJDK developers are very receptive to the idea, were already considering implementing something similar to our SecurityManager rule, and have stated the lessons from the empirical study we shared with in an early manuscript of this paper are valuable to their efforts.

[9] `Class.getProtectionDomain()`

[10] `RuntimePermission(■getProtectionDomain■)`

The SecurityManager rule is enforced by monitoring every read from and write to the `security` field of the `System` class: This field stores the security manager that is used by protected code. The agent implements the read and write monitors by respectively registering `FieldAccess and FieldModification` events for the field. Typically the field is accessed via `System.getSecurityManager()` and modified using `System.setSecurityManager()`, but we must monitor the field instead of instrumenting these methods to detect type confusion attacks.

The agent stores a shadow copy of the application's most recent security manager to have a trusted copy of the manager that can be used to check for rule violations. In a typical deployment, the agent is loaded by a JVM before the hosted Java application's code has begun executing. Even in the typical case, when a security manager is set on the command line that runs the application, the initial security manager would not be caught by the modification event because the write happens before the agent is loaded. To solve this problem, the shadow copy is first initialized by calling `System.getSecurityManager()` when a JVM loads the agent. After this point, the shadow copy is only updated by the modification event, which receives the new manager as a parameter from JVMTI whenever the event is triggered.

Modification events are used to detect any change to a self-protecting security manager. When the field is written, the agent checks the shadow copy of the manager. Assuming the shadow copy is `null`, the agent knows the manager is being set for the first time and checks to see if the new manager is self-protecting. If the manager is self-protecting the agent simply updates the shadow copy. Otherwise the agent stops monitoring the application because the rule does not apply in the presence of a defenseless manager.

Access events are used to detect type confusion attacks against the manager. The modification event we register will not be triggered when the manager is changed due to a type confusion attack. When a type confusion attack is used to masquerade a malicious class as the `System` class, the malicious copy will have different internal JVM identifiers for the class itself and its methods. Even given these differences, updating a field in one version of the class updates the value the JVM stores for the field in both classes because `System` is static and both classes appear to have the same type. The modification and access events are registered for specific field and class identifiers, thus the events are not triggered for operations on the malicious version. We leverage the mismatch this causes between the set security manager and our shadow copy by checking to see if the manager that is read in the access event has the same internal JVM reference as our shadow copy. When the two references do not match, the manager has been changed by a malicious class masquerading as `System`. Type confusion attacks may also be used to masquerade a class as a privileged class loader to elevate the privileges of a payload class that disables the manager; this scenario is detected by the modification event.

7.2 Effectiveness at Fortifying the Sandbox

We performed an experiment to evaluate how effective our rules are at blocking exploits that disable the sandbox. In our experiment, we ran Java 7 exploits for the browser from Metasploit 4.10.0 on 64-bit Windows 7 against the initial release of version 7 of the JRE. This version of Metasploit contains twelve applets that are intended to exploit JRE 7 or earlier, but two did not successfully run due to Java exceptions we did not debug. Metasploit contains many Java exploits outside of the subset we used, but the excluded exploits either only work against long obsolete versions of the JRE or are not well positioned to be used in drive-by downloads.

Table 4: Effectiveness test results.

CVE-ID	Privilege Escalation Monitor	Both Monitors
2011-3544	Attack Succeeded	Attack Blocked
2012-0507	Attack Blocked	Attack Blocked
2012-4681	Attack Succeeded	Attack Blocked
2012-5076	Attack Succeeded	Attack Blocked
2013-0422	Attack Blocked	Attack Blocked
2013-0431	Attack Blocked	Attack Blocked
2013-1488	Attack Succeeded	Attack Blocked
2013-2423	Attack Succeeded	Attack Blocked
2013-2460	Attack Blocked	Attack Blocked
2013-2465	Attack Succeeded	Attack Blocked

We ran the ten exploits in our set under the following conditions: (1) without the agent, (2) with the agent but only enforcing the Privilege Escalation rule, and (3) while enforcing both rules. We ran these conditions to respectively: (1) establish that the exploits succeed against our JRE, (2) test how effective the Privilege Escalation rule is without the security manager rule, and (3) evaluate how effective the agent is in the strictest configuration. Overall, all ten of the exploits succeed against our JRE without the agent. Four were stopped by the Privilege Escalation rule. All ten were stopped when both rules were enforced. The exploits that were not stopped by the Privilege Escalation rule were either type confusion exploits or exploits that did not need to elevate the privileges of the payload class. The payload class does not need elevated privileges when it can directly access a privileged class to exploit. Table 4 summarizes our results using the specific CVEs each exploit targeted.

7.3 Related Work

Our rules increase the security of the sandbox by effectively removing unnecessary features. Prior work has taken a different approach, instead focusing on re-implementing the Java sandbox or adding to the sandbox to increase security. Cappos et al. created a new sandbox structure. They implemented a security isolated kernel to separate sandboxed applications from the main system [32]. They validated this structure by translating past Java CVEs into exploits for the new kernel. Provos et al. describe a method of separating privileges to reduce privilege escalation [43]. Their approach is partially implemented in the Java security model. Li and Srisa-an extended the Java sandbox by providing extra protection for JNI calls [33]. Their implementation, Quarantine, separates JNI accessible objects to a heap which contains extra protection mechanisms. The performance of their mechanism is also measured using DaCapo. Siefers et al. created a tool, Robusta, which separates JNI code into another sandbox [34]. Sun and Tan extend the Robusta technique to be JVM independent [35].

Java applets are the most common ways to transmit Java exploits. Detectors have been created to identify drive-by downloads in JavaScript [36], and in Adobe Flash [37]. Helmer et al. used machine learning to identify malicious applets [38]. Their approach monitored system call traces to identify malicious behavior after execution. However, this approach is entirely reactive. Our approach terminates exploits when they attempt to break out of the sandbox, before the exploit performs its payload. Schlumberger et al. used machine learning and static analysis to identify common exploit features in malicious applets [39]. Blasing et al. used static analysis and dynamic analysis of sandboxed executions to detect malicious Android applications [44]. Unlike these automated approaches, our rules shows that unique mitigation strategies can be created with a better understanding of how applications interact with the sandbox.

7.4 Limitations

Neither of these rules will stop all Java exploits. While the rules catch all of the exploits in our set, some Java vulnerabilities can be exploited to cause significant damage without disabling the security manager. For example, our rules will not detect type confusion exploits that mimic privileged classes to perform their operations directly. However, our rules substantially improve Java sandbox security, and future work will be able to build upon these results to create mitigation techniques for additional types of exploits.

8. CONCLUSION

Our study of Java sandbox usage in open-source applications found that the majority of studied applications do not change the security manager. Some of the remaining applications use the security manager only for non-security purposes. The final set of applications use the sandbox for security and either initialize a self-protecting security manager and never modify it or set a defenseless manager and modify it at run time.

These findings, in combination with our analysis of recent Java exploits, enabled us to build two security monitors which together successfully defeated Metasploit's applet exploits. Some of the studied applications used the security manager to prevent third party components from calling `System.exit()`. More generally, frameworks often need to enforce constraints on plugins (e.g. to ensure non-interference). This suggests that Java should provide a simpler, alternative mechanism for constraining access to global resources. This is supported by our findings that show developers attempting to make non-trivial use of the sandbox often do so incorrectly. One intriguing possibility is to allow programmers to strengthen the policy temporarily (e.g. by adding a permission).

We indirectly observed many developers struggling to understand and use the security manager for any purpose. This is perhaps why there were only 47 applications in our sample. Some developers seemed to misunderstand the interaction between policy files and the security manager that enforces the policy. Other developers appear confused about how permissions work. In particular, they do not realize that restricting just one permission but allowing all others enables a *defenseless* sandbox. In general, sandbox-defeating permissions should be packaged and segregated to prevent accidental creation of defenseless sandboxes. More generally, some developers appear to believe the sandbox functions as a blacklist when, in reality, it is a whitelist. These observations suggest that more resources—tool support, improved documentation, or better error messages—should be dedicated to helping developers correctly use the sandbox.

9. REFERENCES

[1] 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*, pp. 103–112, 1997.

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

[3] L. Garber, "Have Java's Security Issues Gotten out of Hand?," in *2012 IEEE Technology News*, pp. 18–21, 2012.

[4] A. Singh and S. Kapoor, "Get Set Null Java Security." `http://www.fireeye.com/blog/technical/2013/06/get-set-null-java-security.html`, June 2013.

[5] D. Svoboda, "Anatomy of Java Exploits." `http://www.cert.org/blogs/certcc/post.cfm?EntryID=136`.

[6] A. Gowdiak, "Security Vulnerabilities in Java SE," Tech. Rep. SE-2012-01 Project, Security Explorations, 2012.

[7] J. W. Oh, "Recent Java exploitation trends and malware," Tech. Rep. BH-US-12, Black Hat, 2012.

[8] IBM Security Systems, "IBM X-Force threat intelligence report." `http://www.ibm.com/security/xforce/`, February 2014.

[9] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *Asia Pacific Software Engineering Conference (APSEC)*, pp. 336–345, Dec. 2010.

[10] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.

[11] "Permissions in the JDK." `http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html`, 2014.

[12] "Default Policy Implementation and Policy File Syntax." `http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html`.

[13] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of Functional Programming*, vol. 15, pp. 131–177, Mar. 2005.

[14] F. Besson, T. Blanc, C. Fournet, and A. Gordon, "From stack inspection to access control: A security analysis for libraries," in *Computer Security Foundations Workshop*, pp. 61–75, June 2004.

[15] D. S. Wallach and E. W. Felten, "Understanding Java Stack Inspection," in *IEEE Symposium on Security and Privacy*, pp. 52–63, 1998.

[16] Erlingsson and F. Schneider, "IRM Enforcement of Java Stack Inspection," in *IEEE Symposium on Security and Privacy*, pp. 246–255, 2000.

[17] C. Fournet and A. D. Gordon, "Stack Inspection: Theory and Variants," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 307–318, 2002.

[18] M. Pistoia, A. Banerjee, and D. Naumann, "Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model," in *IEEE Symposium on Security and Privacy*, pp. 149–163, 2007.

[19] T. Zhao and J. Boyland, "Type annotations to improve stack-based access control," in *Computer Security Foundations Workshop*, pp. 197–210, June 2005.

[20] "Vulnerability Summary for CVE-2012-0507." `http://web.nvd.nist.gulnerabilitySummaryforCVE-2012-468ov/view/vuln/detail?vulnId=CVE-2012-0507`, June 2012.

[21] N. Hardy, "The Confused Deputy: (or Why Capabilities Might Have Been Invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988.

[22] "Vulnerability Summary for CVE-2012-4681." `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681`, Oct. 2013.

[23] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*. SEI Series in Software Engineering, Addison-Wesley Professional, 1st ed., Sept. 2011.

[24] D. Svoboda and Y. Toda, "Anatomy of Another Java Zero-Day Exploit." `https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=2120`, Sept. 2014.

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 38–49, ACM, 2012.

[26] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking SAML: Be whoever you want to be," in *USENIX Security*, pp. 21–21, 2012.

[27] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *USENIX Security*, 2014.

[28] L. Gong, "Java security: a ten year retrospective," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 395–405, 2009.

[29] "Java Virtual Machine Tool Interface." `https://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/`.

[30] "PrivilegedAction JavaDoc." `http://docs.oracle.com/javase/7/docs/api/java/security/PrivilegedAction.html`.

[31] "Java Web Start." `http://www.oracle.com/technetwork/java/javase/javawebstart/index.html`.

[32] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson, "Retaining sandbox containment despite bugs in privileged memory-safe code," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 212–223, ACM, 2010.

[33] D. Li and W. Srisa-an, "Quarantine: A Framework to Mitigate Memory Errors in JNI Applications," in *Conference on Principles and Practice of Programming in Java (PPPJ)*, pp. 1–10, 2011.

[34] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the Native Beast of the JVM," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 201–211, 2010.

[35] M. Sun and G. Tan, "JVM-Portable Sandboxing of Java's Native Libraries," in *European Symposium on Research in Computer Security (ESORICS)*, pp. 842–858, 2012.

[36] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code," in *International World Wide Web Conference (WWW)*, pp. 281–290, 2010.

[37] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and Detecting Malicious Flash Advertisements," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 363–372, 2009.

[38] G. Helmer, J. Wong, and S. Madaka, "Anomalous Intrusion Detection System for Hostile Java Applets," *J. Syst. Softw.*, vol. 55, pp. 273–286, Jan. 2001.

[39] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead Analysis and Detection of Malicious Java Applets," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 249–257, 2012.

[40] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pp. 169–190, Oct. 2006.

[41] "Applet usage statistics." `http://trends.builtwith.com/docinfo/Applet`.

[42] "Java Control Panel." `https://docs.oracle.com/javase/7/docs/technotes/guides/jweb/jcp/jcp.html#java`.

[43] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," in *USENIX Security*, 2003.

[44] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Conference on Malicious and Unwanted Software (MALWARE)*, pp. 55–62, 2010.