Fortifying the Java Sandbox

Zack Coker, Michael Maass, Tianyuan Ding, and Joshua Sunshine
School of Computer Science
Carnegie Mellon Univeristy, Pittsburgh, PA
{zfc, mmaass}@cs.cmu.edu, tding@andrew.cmu.edu, 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 maliciousness. However, many recent exploits have successfully escaped the sandbox, thereby enabling attackers to infect countless Java hosts. To prevent future exploits it is essential to distinguish patterns of malicious use from patterns of benign use. 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 do both routinely. We used these results to develop two runtime monitors, one that prevents security manager modification and one that prevents privilege escalation. The privilege escalation monitor stops four of ten Metasploit Java exploits with negligible overhead. The combination of both monitors stops all ten exploits, but incurs very significant overhead, suggesting that the best application is to risky settings like running applets.**

I. Introduction

The Java Runtime Environment (JRE) is widely installed on user endpoints and it executes external code in the form of applets. These facts, combined with the hundreds of recently discovered vulnerabilities in Java, including zero-day vulnerabilities (e.g. CVE-2013-0422), have made Java an extremely popular exploit vector (see Figure 1). Attackers typically lure users to websites containing hidden, yet 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.

Java was designed to safely execute untrusted code and safely isolate components from each other in a sandbox so that the application and the host machine are protected from malicious behavior. However, the exploits cited above show that there is substantial room for improvement. Past investigations of Java exploits have shown Java malware commonly alters the sandbox's settings [1]. Typically, exploits disable the security manager, the component of the sandbox responsible for enforcing the security policy [2], [3], [4], [5]. It seems plausible that benign applications interact with the security manager differently. If true, this difference can be exploited to prevent future attacks. To investigate this further, 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[7] and the GitHub repository that make use of the security manager. We analyzed the resulting 47 projects using a custom FindBugs [8]plugin to isolate code involved in the initialization or modification of the security manager. We then manually categorized 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 categorizations of code were accurate at runtime.

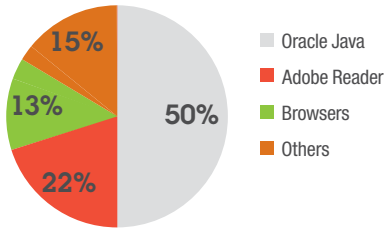

Figure 1. Most targeted applications in the enterprise, according to a Dec. 2013 survey of Trusteer customers [6].

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

Based on our analysis of benign and malicious applications, we implemented two runtime monitors. The first monitor prevents privilege escalation by preventing restricted classes inside a sandbox from loading classes with fewer restrictions. The second monitor, prevents changes to the sandbox when a self-protecting security manager is initialized. We evaluated the effectiveness of our monitors against the 10 applets in Metasploit 4.10.0 that successfully exploit unpatched versions of Java 7. The privilege escalation monitor detected and stopped four of the ten exploits, while using both monitors together detected and stopped all ten exploits.

We evaluated the performance of our monitors using the DaCapo benchmark suite[9]. The privilege escalation monitor

resulted in negligible overhead of X% which is low enough to enable monitoring of any Java application. The security manager monitor is implemented as a JVMTI agent and it monitors a static field, which unfortunately disables just-in-time compilation (JIT). Therefore, the security manager monitor resulted in substantial overhead of XXX%, which is unacceptable for most Java applications. However, the greatest threat comes from Java applets, and many of these are not performance intensive (e.g. they are web forms), so it may be reasonable to enable the security manager monitor only for untrusted applets and use only the privilege escalation monitor for all other applications.

The contributions of this papers are as follows:

- An empirical study of Java sandbox usage in benign, open-source applications (Section IV).
- An analysis of privilege escalation in the Java security model and recent Java exploits (Section III).
- Two novel rules for distinguishing between benign and malicious Java programs (Section V).
- Implementations of the two rules as runtime monitors, with accompanying security and performance evaluations (SectionVI).

II. Background on the Java Sandbox

The Java sandbox was designed to safely execute code from untrusted sources. Figure 2 summarizes the components of the sandbox that are relevant to this work. 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 is used to indicate the origin of the code and to associate the class with a protection domain. Protection domains segment the classes of an application into different 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 [10]). Policies written in the Java policy language[1] explicitly define permission sets and assign code sources to each set. By default, applications executed from the local file system are run without a sandbox, while all other applications are run inside a restrictive sandbox. This prevents the execution of malicious operations on the host system by applications from the network or other untrusted sources.

Even if a policy is defined, the policy 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 security manager is queried by the protected method to determine if the operation should be allowed. For example, if an application attempts to write to a file

[1]http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html

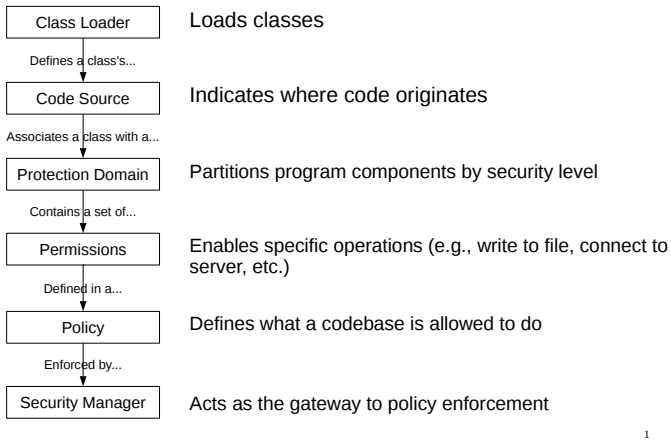

Figure 2. A high-level summary of the Java sandbox as it pertains to this work.

(e.g `java.io.FileOutputStream`) inside a sandbox, the class that performs the write will check with the security manager to ensure a write to that file is allowed. Due to the fact that it is up to the protected code to initiate the check, missing checks are a common source of Java vulnerabilities.

To perform a permission check, the security manager walks the stack to ensure each class in the current stack frame has the required permission. However, the stack walking can be stopped before the entire frame has been walked by a privileged class, when the privileged class specifically wraps code inside a `doPrivileged` call. This allows for 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 [11], [12], [13], [14], [15], [16], [17].

Java provides flexibility when setting up a sandbox, allowing a sandbox to be set at any time during the execution of an application, or in many cases, before an application is started. In the default case for web applets and applications that use Java Web Start, a *self-protecting* security manager is set before the application is loaded 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 the settings of the sandbox. A security manager can also be *defenseless*, which in this case is the exact opposite of self-protecting. A defenseless manager does little to improve the security of a constrained application or the system it executes on. However, we show in section IV-C that some benign applications have found interesting uses for defenseless managers. Table I summarizes the set of permissions used to distinguish between self-protecting and defenseless security managers. A security manager that enforces a policy that contains even one of the listed permissions is defenseless. A subset of the permissions in this list were identified in [4].

Table I
A SECURITY MANAGER ENFORCING A POLICY THAT CONTAINS ANY PERMISSION IN THIS LIST IS DEFENSELESS.
*ANY COMBINATION OF WRITE OR EXECUTE IN THIS PERMISSION ENSURES THE MANAGER IS DEFENSELESS.

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

```
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() throws Exception {
        // Disable sandbox
        System.setSecurityManager(null);
    }

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

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

III. EXPLOITING JAVA CODE

Malicious drive-by downloads using Java applets as the vector were widely reported between 2011 and 2013. While the Java sandbox should prevent malicious applets from executing their payloads, vulnerabilities in the Java Runtime Environment (JRE) were leveraged by exploits to set the security manager to `null`. Setting the security manager to `null` disables the sandbox, which allows previously constrained classes to perform any operation that the JRE itself has the privileges to perform. Figure 3 shows a typical payload.

Some Java exploits use type confusion to bypass the sandbox. A type confusion vulnerability is exploited by breaking type safety, thus 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 will craft objects that either (1) point to the `System` class to cause any operation they perform to happen on the real `System` class, thus allowing them to directly alter the field where the security manager is stored or (2) act as if they have the same type as a privileged class loader to load a payload class with all permissions (see CVE-2012-0507 [18]).

Another prominent subclass of Java exploits take advantage of a confused deputy vulnerability [19], which is a subset of privilege escalation. In the case of a confused deputy, the exploit often convinces a class with access to a vulnerable privileged class to return a reference to the latter. The returned privileged classes often contain a vulnerability in the form of a missing security check, where the class should consult with the security manager before performing some operation but doesn't. In some cases the privileged class may be directly accessible, but this is quite rare and typically the fault of a vulnerable third-party library because 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, the class is usually tricked into executing code that disables the sandbox (see CVE-2012-4681 [20]).

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 Java provides applications less-privileged access to. For example, many reflection operations are implemented in the `sun.reflect` package, which has all permissions, but 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.

To load a privileged class, a privileged class loader must be used, thus a class should typically 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 them to execute operations, the sandbox in its current form would serve little purpose. We discuss later 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 was developed while strictly following "The CERT Oracle Secure Coding Standard for Java" [21]. For example, Svoboda [3], [22] 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. In the rest of this paper

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

we concern ourselves with ways to fortify the Java sandbox without breaking backwards compatibility and not with the specifics of particular exploits.

IV. Security Manager Study

Our intent is to pro-actively stop exploits that disable the Java sandbox. We focus our efforts on the security manager as it is the means by which applications interact with the sandbox. To successfully stop even 0-day exploits, we must understand which operations both exploits 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, we can exclude the operations that exploits depend on that are not of use to benign applications. This outcome could effectively narrow the range of possible operations on the manager to stop exploits while achieving backwards compatibility with benign applications. Additionally, 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 results of an empirical study that validated this strategy.

A. Methodology

As discussed in 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 the following hypotheses, roughly categorized by the strength of the mitigation that is possible if the hypothesis is true:

Weak Hypothesis: *Benign applications do not disable the security manager.* If this hypothesis is true, exploits can be differentiated from benign applications by any attempt to disable the current security manager. While this mitigation would be easy to implement, 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`.

Moderate Hypothesis: *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 mitigation would require differentiating between changes which weaken the sandbox and those that do not. Classifying changes in this manner 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?

Strong Hypothesis: *Benign applications do not change the sandbox if a self-protecting security manager has been set*. If true, it is possible to implement a mitigation strategy whereby any change to a security manager that is enforcing a strict policy (as defined in section II) is disallowed. To implement this mitigation a runtime monitor must determine if a security manager is self-protecting at the time the manager is set, which can be easily achieved. While this mitigation has the same outcome as the mitigation enabled by successful validation of the moderate hypothesis, this mitigation is significantly easier to implement and is therefore stronger.

Ideal Hypothesis: *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.

Our empirical analysis used applications from the Qualitas Corpus (QC) [7] 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*. Assuming 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 attempted to compile each of the 29 included applications and updated those where problems arose with modern build tools to the newest version. Section IV-B lists the versions of the applications used in this study.

We performed a similar process using the GitHub search feature configured to search through Java files for the same keyword. Initially, we extracted the top 6 applications from the search results, but we came to find this filtering method was producing a high false positive rate: 4 out of 6 of the applications didn't actually use the `SecurityManager` class (e.g. the keyword appeared in a code comment). To counteract these false positives, we refined our search to more precisely include applications that set a manager by using the keyword *System.setSecurityManager(*. We selected the top 7 applications from these search results while keeping the true positives from the previous search. To ensure we included applications that disable the security manager, we repeated this process using the keyword *System.setSecurityManager(null)*. From this starting point of 20 applications, we excluded 2 that were already covered in the Qualitas Corpus and a Ruby application that mistakenly made it into the set because it contained Java files. We always downloaded the latest commit of each application to ensure the GitHub dataset reflected their most current versions.

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 [8] plugin that uses a dataflow analysis to determine where `System.setSecurityManager()` is called, as well as the lines of code where the method's arguments were initialized. We also created a dynamic analysis tool using the Java Virtual

Machine Tool Interface (JVMTI)[3]. 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 particular field holds the current security manager object for the application, which is used throughout the application's execution to ensure that code has the correct permissions to perform protected operations. 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 steps listed here:

1) The reviewer ran grep on all Java source files in the application to output the lines which contain the keyword *SecurityManager* and the 5 lines before and after these lines.
2) When it was clear from the grep output that the keyword was used in comments or in ways that were unrelated to the security manage class, the reviewer labeled the application as a false positive.
3) For true positives that compiled, the reviewer ran FindBugs on the application with only our plugin enabled.
4) The reviewer manually inspected code specified in the FindBugs findings, starting with the line where the manager was set and tracing the code back to the various locations the findings said potential security managers were initialized.
5) The reviewer manually inspected all of the lines mentioned in the grep results from step 1 to see how the application interacted with the sandbox.
6) For true positives that compiled and effected the security manager during the execution of the application, the application was executed, while being monitored by our dynamic analysis tool, using parameters and actions the reviewer determined in steps 4 and 5 effect the security manager. For example, we often learned in earlier steps that the manager was only effected if the user ran the program from the command line with a particular parameter or used a specific feature of the application. This step verified the conclusions from previous steps.
7) Finally, the reviewer summarized the operations the application performed on the security manager with an emphasis on points that support or reject each hypothesis.

To ensure the reviewers understood the analysis steps and produced consistent results, we undertook a pilot study where each reviewer independently inspected the same 6 applications. This pilot study was invaluable in ensuring the inspections were performed consistently because one of the reviewers played no role in creating the tools and was therefore less informed about what to expect than our more experienced reviewer.

[3]http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/

Table II
APPLICATIONS STUDIED

Application Name	Description	Repo
(Apache) Ant	Java Project Builder	Qualitas
(Apache) Batik	SVG Image Toolkit	Qualitas
C-JDBC	DB Clustering Middleware	Qualitas
Compiere	Business Tools	Qualitas
(Apache) Derby	Relational Database	Qualitas
DrJava	IDE	Qualitas
Eclipse	IDE	Qualitas
FreeMind	Mind-Mapping Tool	Qualitas
Galleon	Media Server	Qualitas
(Apache) Hadoop	Distributed Computing Frwk.	Qualitas
Hibernate	Obj.-Rel. Mapping Tool	Qualitas
HyperSQL	SQL Relational Database	Qualitas
JBoss	Application Middleware	Qualitas
JRuby	Ruby Interpreter	Qualitas
(Apache) Lucene	Search Software	Qualitas
(Apache) MyFaces	Server Software	Qualitas
NekoHTML	HTML Parser	Qualitas
Netbeans	IDE	Qualitas
OpenJMS	Messaging Service	Qualitas
Quartz	Job Scheduler	Qualitas
QuickServer	TCP Server Framework	Qualitas
Spring Framework	Web Development Library	Qualitas
(Apache) Struts	Web Development Library	Qualitas
(Apache) Tapestry	Web Development Library	Qualitas
(Apache) Tomcat	Web Server	Qualitas
Vuze	File Sharing Application	Qualitas
Weka	Machine Learning Algs.	Qualitas
(Apache) Xalan	XML Transforming Library	Qualitas
(Apache) Xerces	XML Parsing Library	Qualitas
AspectJ	Java Extension	Github
DemoPermissions	Spring Extension	Github
driveddoc	Application Connector	Github
FileManagerFtpHttpServer	FTP Server	Github
Gjman	Development Toolkit	Github
IntelliJ IDEA	IDE	Github
Jmin	Lightweight JDK	Github
MCVersion-Control	Minecraft Version Changer	Github
NGOMS	Business Tools	Github
oxygen-libcore	Android Dev. Lib.	Github
refact4j	Meta-model Prog. Frwk.	Github
Security-Manager	Alternate 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

B. The Security Manager Dataset

The Qualitas Corpus is a curated collection of open source Java applications for use in reproducible software studies. We investigated the sandbox usage of 29 applications from QC version 20130901 that are used in this study. 11 of the 29 applications are developed by the Apache Software Foundation (ASF), which may increase the homogeneity of their operations on the manager.

While QC provides a strong starting point for the construction of a dataset for this study, their inclusion criteria[4] lends to the inclusion of large, popular applications and frameworks. Given this point and the emphasis on ASF applications in our filtered set, we chose to diversify our dataset by turning to

[4]http://qualitascorpus.com/docs/criteria.html

Table III
CLASSIFICATION OF APPLICATION INTERACTIONS WITH THE SECURITY MANAGER

Type of Interaction	Qualitas	GitHub	Total
1. Set a manager without later changing it	6	1	7
2. Change a set security manager	5	3	8
3. Interact with manager in production code	10	3	13
4. Interact with manager only in unit tests	3	5	8
5. Do not interact (false positive)	5	5	10

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

Figure 4. The only location in the Weka code where interaction with a security manager occurs.

GitHub. The full list of applications studied are listed in Table II.

C. 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 III shows a breakdown of how each application in our dataset was categorized. The number for each category type in the table corresponds to the number in the previous list and 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 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, didn't actually set a security manager, but did set the security manager to null as discussed in the section below titled "Reducing Web Application Development Complexity".

Type 2 applications are of particular interest in validating the hypotheses because they change a set security manager at some point later in the application's execution. In other words, they potentially falsify the weak, moderate, and ideal hypotheses. Due to their effect on our hypotheses, applications of this type are discussed in detail below.

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[5] to ensure the application works if constrained. Similarly, type 4 applications contain code in unit test that ensure the application works correctly if sandboxed or that set a manager themselves, but 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 whose type does 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.

1) Evaluation of our hypotheses: We only require one counterexample to falsify a hypothesis from section IV. This section summarizes how our hypotheses held up against the results of this study.

Weak Hypothesis: *Benign applications do not disable the security manager.* The investigation determined that some benign applications 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 IV-C2. However, some of these applications turned off the sandbox temporarily to update the imposed security policy.

Moderate Hypothesis: *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.

Strong Hypothesis: *Benign applications do not change the security manager if a self-protecting security manager has been set.* This hypothesis was supported by both datasets. When false positives are excluded, 19 out of 24 true positives in the Qualitas dataset and 9 out of 12 true positives in the GitHub dataset did not change a set security manager or the policy it enforced during execution.

[5]http://docs.oracle.com/javase/7/docs/api/java/security/PrivilegedAction.html

```
691 System.setSecurityManager(new AntSecurityManager(
        originalSM, Thread.currentThread()));
692 ...

703 getCurrentProject().executeTargets(targets); \\Note:
        Ant is executed on this line
704 ...

721 finally {
722 ...

725     if (System.getSecurityManager() instanceof
            AntSecurityManager) {
726         System.setSecurityManager(originalSM);
727     }
```

Figure 5. This code snippet from Eclipse shows how it uses the manager to prevent Ant from terminating the JVM when an unrecoverable error occurs.

```
703 public static void apply() {
704     final SecurityManager securityManager = new
            SecurityManager() {
705         public void checkPermission(Permission
                permission) {
706             if(permission.getName().startsWith("exitVM"
                    )) {
707                 throw new Exception();
708             }
709         }
710     };
711     System.setSecurityManager(securityManager);
712 }
713 public static void unapply() {
714     System.setSecurityManager(null);
715 }
```

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

Ideal Hypothesis: *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.

2) Non-security uses of the Sandbox: We found several cases where applications use the sandbox in ways that were not intended to increase the security of the system. 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.

Enforcing Architectural Constraints : It is common for Java applications to 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.

This case appears in Eclipse, which uses Ant as a library. Ant kills the JVM when an unrecoverable error condition occurs to terminate execution of the build script it is running. However, Eclipse wants to continue executing and to report an error to the user when Ant runs into a error condition. 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. The code references a blog post[6] that appears to be the origin of this solution. 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 is probably the best solution available in Java at the moment, it is likely to cause problems when applications are also using the sandbox for security purposes. The technique requires the application to dynamically change the security manager, which 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.

Reducing Web Application Development Complexity: During our investigation, we found that the Java security policies for web applications (applets and applications launched via Java Web Start [7]) created complexity for certain applications. In the latest versions of the JRE at the time of this paper (1.7.72 and 1.8.25), Java requires that users agree that a Java web application originating from a source other than the host machine should be executed. By default, Java executes the application inside a restrictive sandbox that severely limits the operations the application can perform. Inside the restrictive sandbox, applications cannot perform operations such as accessing files on the local filesystem, retrieving resources from any third party server, and changing the security manager.

For some applications, the restrictions imposed by the restrictive sandbox may prevent required behavior. Thus, Java provides a way to run applications outside 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

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

[7]http://www.oracle.com/technetwork/java/javase/javawebstart/index.html

```
22 /**
23 * The launcher to start eclipse using webstart. To use
       this launcher, the client
24 * must accept to give all security permissions.
25 ...

55 public static void main(String[] args) {
56 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.

should run outside of the sandbox. Before running this kind of application outside of the sandbox, the use must accept two prompts: a pop-up to run the application and second pop-up 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 restrictive sandbox and then avoid using restricted permissions. Thus, the application would only be able to execute a subset of the full application. This option requires more development effort because the application is essentially implemented twice, once as the restricted version and again as the full version. The other option does not allow the application to execute the application inside a restrictive sandbox. This option forces the user to run the application outside the sandbox, since the application would not work otherwise. 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 believe developers did not allow the application to terminate farther into the application's execution to prevent the user from misdiagnosing a permission violation for another error. During the investigation, we found two applications that used this method: 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.

3) Changing the Security Manager for Security Purposes: We found applications that set a security manager then either explicitly change it at some later point or allow the user to change it. 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 was designed to allow users to optionally constrain the execution of an an

```
156 public void enforceSecurity(boolean enforce){
157 SecurityManager sm = System.getSecurityManager();
158
159 if (sm != null && sm != lastSecurityManagerInstalled){
160     ...

163     throw new SecurityException
164         (Messages.getString(
                EXCEPTION_ALIEN_SECURITY_MANAGER));
165 }
166 if (enforce) {
167     ...

173     installSecurityManager();
174 } else {
175     if (sm != null) {
176         System.setSecurityManager(null);
177         lastSecurityManagerInstalled = null;
178         ...
```

Figure 8. Security manager interactions in Batik.

application that uses the Batik library. The method takes one parameter that acts as a switch to turn the sandbox on or off. Batik throws an exception if the user wants to toggle the sandbox while a non-Batik manager is set. The download page on the Batik website shows several examples of how to use the library. Two of the examples show ways to 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 can be set to call `enforceSecurity` with a true argument the first time and a false argument the second time, which enables then disables the sandbox. 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.

As mentioned earlier, Ant, Freemind, and Netbeans explicitly set and then change the current 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 set of permissions. The user specified set of permissions is defined in the permissions element of an Ant build file. Inside the permissions element, a user can explicitly grant and revoke permissions available to the Java code. 9 shows the first example on the Ant Permissions website[8]. In this example, the contents of the `grant` element provide the application all permissions. Then the contents of the revoke element restricts the application from using all property permissions. This permissions snippet means that the application can perform all actions except for those that require `PropertyPermissions`. This snippet also means that Ant will set a defenseless security manager. Malicious code can easily bypass the restrictions in this example by executing the line `System.setSecurityManager(null)`, which turns off the sandbox. Once the sandbox has been turned off, the malicious code can perform all actions which require `PropertyPermissions`. The Ant Permissions website does not mention that an implementation of this example

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

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

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

```
 98  public synchronized void setSecurityManager() throws
         BuildException {
 99    origSm = System.getSecurityManager();
100    init();
101    System.setSecurityManager(new MySM());
102    active = true;
103  }
104
105  /**
106    * Initializes the list of granted permissions, checks
           the list of revoked permissions.
107    */     private void init() throws BuildException {
108    ...

169  public synchronized void restoreSecurityManager() {
170    active = false;
171    System.setSecurityManager(origSm);
172  }
```

Figure 10. A code snippet showing Ant's custom security manager architecture, used to enable build scripts run Java code under a specific set of permissions unique to the instantiated code.

would be vulnerable. The site also does not mention that allowing certain permissions will potentially enable the restricted application to bypass any user specified restrictions.

To provide the user the ability to grant and revoke permissions, Ant uses the code section from Permissions.java shown in 10. When Ant is about to execute the restricted application, Ant calls the `setSecurityManager` method of the Permissions class. 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, the security manager allows all permission checks to succeed, making it a defenseless security manager. Once the `setSecurityManager` method changes altered to true, `MySM` enforces the specified permissions.

When Ant has finished executing the application, it calls the method `restoreSecurityManager` to remove the security manager. This method sets active to false, which turns off all permission checks performed by `MySM`. Once the permission checks of the current security manager are turned off, the method restores the original security manager.

With this implementation, Ant catches applications that perform actions restricted by the user, since the security manager is active when the application is executing. 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. In both of these cases a defenseless security manager is required, otherwise the application would not be able to change the current security manager. However, this technique may cause problems if Ant or Netbeans is used in a security critical setting, since a security critical setting would need 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 be in line with the way the sandbox is intended to be used and would be more clearly correct and trustworthy while allowing Ant and Netbeans to run inside of a self-protecting sandbox.

```
133 public void checkPermission(Permission pPerm, Object
        pContext) {
134   if(mFinalSecurityManager == null) return;
135   mFinalSecurityManager.checkPermission(pPerm, pContext)
          ;
136 }
```

Figure 11. This code snippet shows how Freemind's custom security manager selectively enforces permissions.

```
30 /**
31 * By default, everything is allowed.
32 * But you can install a different security controller
       once,
33 * until you install it again. Thus, the code executed in
34 * between is securely controlled by that different
       security manager.
35 * Moreover, only by double registering the manager is
       removed. So, no
36 * malicious code can remove the active security manager.
37 *
38 * @author foltin
39 *
40 */
41 public void setFinalSecurityManager(SecurityManager
       pFinalSecurityManager) {
42     if(pFinalSecurityManager == mFinalSecurityManager)
           {
43   mFinalSecurityManager = null;
44       return;
45   }
46   if(mFinalSecurityManager != null) {
47   throw new SecurityException("There is a
         SecurityManager installed already.");
48   }
49   mFinalSecurityManager = pFinalSecurityManager;
50 }
```

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

Freemind 0.9.0 tried to solve a similar problem and ended up illustrating the dangers of a defenseless manager. Freemind is a mind mapping tool that allows users to execute Groovy scripts on a map opened in the editor. The scripts are written by the creator of the mind map. Groovy is a scripting language that is built on top of the JRE, where 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, but while limited by the sandbox that encapsulates the hosting application.

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

```
199     public void checkSecurityAccess(String pTarget) {
200 }
```

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

```
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. Our example exploit that breaks out of the scripting sandbox in Freemind to execute arbitrary code.

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, `pFinalSecurityManager`, that specifies the manager that should 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. Initially, the `setFinalSecurityManager` method sets the field to the method argument. Once set, if the method is called again with a different security manager, a `SecurityException` is thrown, but calling the method with a reference to the same manager the field already contains 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 scripts to create network sockets, access the file-system, or execute programs on the machine if the script is unsigned. The manager in this case explicitly allows all other permissions. Figure 13 shows a sample permission check from the custom manager in ScriptingSecurityManager.java that is set as the proxy manager. `checkSecurityAccess` is a security manager method that checks if the sandbox may be changed. Due to the empty implementation of this method, all checks will succeed as if permission to reconfigure the sandbox was granted. A malicious script can call `System.setSecurityManger(null)` to turn off the sandbox at any point, and then perform the previously restricted actions.

To show that the problem with the current implementation was more complex than fixing this permission check alone, we demonstrated that the custom security manager can easily be removed using reflection. Figure 14 shows an exploit written as a Groovy script for a mind map that turns off the manager. The script gets a reference to the system's manager, then the class of the custom security manager. With the class of the manager, the exploit gets a reference to the proxy manager field. Next, the field's modifier is flipped from private to public before the exploit reflectively nulls it, thus effectively turning off the sandbox. Finally, the exploit creates a file to demonstrate that a "forbidden" operation succeeds.

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.

As a final example of security manager interactions that are for security purposes, we very briefly look at WildFlySecurityManager, a custom security manager that does not cleanly fit into our categories. WildFlySecurityManager allows permission checks to be disabled for classes granted a custom permission called `DO_UNCHECKED_PERMISSION`. This strategy is equivalent to running a privileged action with `doPrivileged`.

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[9], called inspections, to warn users of potentially problematic sections of code. In the inspections for security issues, the IntelliJ editor contains two inspections that directly pertain to interactions with the sandbox: 1) one that highlights calls to `System.setSecurityManager()` and 2) another that highlights the definition of custom security manager classes. The descriptions for these inspections respectively contain warnings that imply that interaction 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 concern with developers interacting with the security manager in these ways is clearly warranted given our findings above.

V. Rules for Fortifying the Sandbox

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

A. Privilege Escalation Rule

The *privilege escalation rule* ensures that a class may not directly load a more privileged class if a self-protecting security manager is set for the application. This rule is violated

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

when the protection domain of a loaded class implies a permission that in not implied in the protection domain that loaded it. Many 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.

B. 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, which many exploits try to ensure will happen.

VI. Mitigations

In section IV we discussed (1) four hypotheses that could lead to Java exploit mitigations if validated and (2) how we went about validating them. We also discussed additional information about how applications use the Java sandbox that we learned while successfully validating the strong hypothesis and the overall results of our empirical analysis of the open source Java landscape. The results included two backwards-compatible rules, discussed in section V, that can be enforced to stop current exploits. In this section we discuss the implementation and evaluation of runtime monitors that implement the privilege escalation and Security Manager rules. We collectively call these monitors the *Java Sandbox Fortifier* (JSF). We evaluated JSF in collaboration with a large aerospace company that is currently working on deploying the tool to workstations that belong to employees that are often the subject of targeted attacks.

A. Implementation Using JVMTI

Prior work has made an effort to prevent exploits in the native libraries used by language runtimes such as Java's [23], [24], [25], [26], and the machine learning community has put some effort into detecting exploits delivered via drive-by-downloads using Java applets and similar technologies [27], [28], [29], [30]. We implemented a tool in JVMTI to proactively stop exploits written directly in the Java programming language to exploit vulnerable Java code[10]. In particular, our tool blocks operations that exploits use without effecting the execution of benign applications.

JVMTI is a native interface used to access JVM operations that are intended to be 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 some configuration specific point in the application's lifecycle. The interface allows an agent to set capabilities that enable the tool to intercept events such as class and thread creation, field access and modification, breakpoints, and much more. After acquiring the necessary capabilities, a JVMTI agent registers callbacks for the events they want to receive. JVMTI provides a rich API, hooks for instrumenting the bytecode of loaded classes, and access to the JNI, all of which can be used to perform nearly any operation on classes, threads, etc. that a tool may want to perform at the time when an event occurs. 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 below.

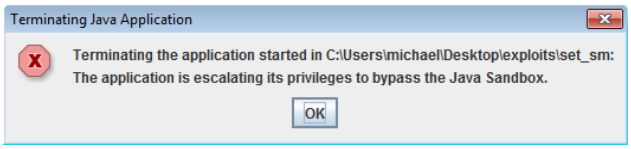

Figure 15. A popup from our agent after it caught an exploit breaking the privilege escalation rule.

Our agent was written in C++. 524 lines of code were required to enforce the privilege escalation rule while 377 lines of code were required for the Security Manager rule when counted using the Linux tool `wc`. This code constitutes the attack surface for our tool because a malicious class could potentially craft information such as class, field, or method names to exploit an issue in the rule enforcement code when the information is passed to the appropriate callback. The risk here is greatly reduced both by the fact that there is little attack surface to inspect and due to the previously cited work that can be applied to our tool. For example, the software-based fault isolation subset of Robusta [25] can be applied to our tool to isolate the effects of an exploit. Using a security kernel for Java similar to Cappo's for Python [23], our tool could be isolated to its own security layer with access only to the information it gets from JVMTI. We did not attempt to apply these solutions because the required tools and code are not publicly available, which would make it difficult, if not impossible, for most people to adopt our tool.

Our agent may be configured to run in enforce or monitor mode. In enforce mode a violation of either rule causes the agent to log the offending behavior and terminate the JVM to which the agent is attached. A popup is shown to the user to let them know why their Java application was terminated when the agent has been configured to show popups (this was made configurable to prevent popups on headless servers). Figure 15 shows an example of a popup displayed after an exploit was caught breaking the privilege escalation rule. In monitor mode the agent logs the offending behavior but leaves the JVM's execution of the application untouched.

[10]Our tool, Java Sandbox Fortifier, is open source and hosted on GitHub at https://github.com/SecurityManagerCodeBase/JavaSandboxFortifier. **REVIEWERS: INSPECTING THIS GITHUB PROJECT MAY REVEAL THE AUTHORS' IDENTITIES.**

Enforcing the Privilege Escalation Rule: The privilege escalation rule is enforced by ensuring that, after a self-protecting security manager has been set, classes do not load or cause the loading of more privileged classes unless the privileged class is in a restricted-access package. *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 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. The latter option would ensure the security manager is defenseless, thus the application would not be protected by the agent (see table I).

We must allow a class to indirectly load a class in a restricted-access package because classes in these packages are often used by JRE classes that an application is allowed to use. For example, many of the classes in the `java.lang.reflect` package are backed by classes in the `sun` package, the latter of which is a restricted-access package that contains the internal implementations for many Java features. However, enforcing this rule prevents exploits from elevating the privileges of their payloads because the payloads are not in restricted-access packages and cannot be in them with default JRE configurations.

To enforce this rule, our agent registers for the `ClassPrepare` event, which allows it 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, but this can not be done by calling the necessary Java methods[11] via the JNI from the agent because the Java calls will be performed with the same permissions as the application the agent is attached to. Most applications where this operation is relevant (i.e. those that have a self-protecting manager) do not have the necessary permission to get a protection domain[12] because it would allow a malicious class to probe the policy to determine which, if any, malicious operations it can perform. Due to the fact that 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 implementation of the agent as of the time of this writing simply checks to see if the loaded class's protection domain has all permissions while the class that caused the loading doesn't. 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 cases we are currently not aware of.

Enforcing the SecurityManager Rule: 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, which is private and static, 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, as discussed later in this section.

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 the agent is loaded by a JVM. 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 will also drop into monitor mode when enforce mode is configured because the rules cannot be enforced for applications that use defenseless managers. We cannot enforce the rules in the presence of a defenseless security manager because enforcement may break the function of benign applications that utilize a defenseless manager, as in several examples in section IV-C. In any case, future modifications are logged as a violation of the rule and trigger the operation relevant to the agents current mode as discussed above.

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 field that stores the manager, the class itself, and its methods even though writing to the field in one version of the class updates the same field in the other version. The

[11] `Class.getProtectionDomain()`
[12] `RuntimePermission("getProtectionDomain")`

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 in the access event by checking to see if the manager that was just read has the same internal JVM reference as our shadow copy. When the two references do not match, the manager has been changed as the result of a malicious class masquerading as the `System` class. 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, but this scenario is detected by the modification event.

B. Overhead

We measured performance overhead using version 9.12-bach of the DaCapo Benchmark Suite [9], a standard set of real-world Java applications used for Java benchmarking. Performance was measured by running DaCapo 5 times using the converge switch *–converge –max-iterations 30 –window 3*. The average for each benchmark is the geometric mean of the last benchmark execution times from the 5 convergence iterations. We ran the benchmarks on an otherwise idle laptop with 8 gigabytes of RAM and an 8 core 64-bit Intel i7-3632QM CPU at 2.2 GHz. We used the 64-bit version of Java version 1.7.0 update 60.

The DaCapo benchmarks do not set a security manager. Thus, to measure JSF's overhead, we used a modified version of the agent that always performs rule checks. This produces the worst-case overhead because the unmodified version of JSF only attempts to monitor for rule violations when a security manager has actually been set. IV contains the performance results from these tests.

Our results indicate that the privilege escalation rule causes minimal slowdown with an average of -0.8% overhead, which we believe is due to variations in the benchmark runs and not the tool. On the other hand, the Security Manager rule produced a significant slowdown, causing applications to run over 45 times slower on average. We investigated the cause of this slowdown and found that read and write monitors on fields in JVMTI cause the program to run without the benefit of the JIT. We measured the benchmarks in interpreted mode to measure how much of this extensive slowdown is the result of the JIT being off. The tests determined that the interpreted applications runtime increased by 14.1 times relative to the JITed version of the application. While we do not break the figures down, the rest of the slowdown is likley caused by the JRE executing less efficient code paths when certain JVMTI capabilities are enabled along with the actual enforcement of our rule.

We also measured performance with a version of the agent that receives the required events for both monitors but does nothing with them—the code that enforces the rules in events was removed. These tests produced results that are nearly identical to the results we received when the rule implementations were in place. This strongly suggests that the significant performance slowdown is caused by JVMTI's implementation of the events we register for to enforce the security manager rule.

The security manager rule slowdown can be largely mitigated by advancing JVMTI implementations, for example, to enable the JIT in all cases. However, JVMTI implementors tend to favor ease of implementation over speed because many uses of JVMTI, outside of research, are not performance critical. We hypothesize that the security manager rule can be implemented with significantly less performance overhead if the rule is built into the JVM. This approach has the added benefit that the rules would become a permanent mitigation for all Java applications. At the time of this writing, we are actively communicating with Java developers to explore the prototype implementation of these rules in OpenJDK.

The user effort required to use JSF is minimal, and several install options are possible which can help counteract the security manager rule's overhead. To use the tool in the default configuration, a user either has to compile the tool using the provided makefile or download a binary version of the tool for their system. Once compiled, JSF can be set to fortify all Java applications on a standard Oracle JVM in two steps: 1) set the `JAVA_TOOL_OPTIONS` environment variable to *-agentpath=path_to_jsf_library* which causes the JVM to attach the agent to every JVM that is started after this point and 2) set the `JSF_HOME` environment variable to point to the directory containing JSF and its configuration files. By default the security manager rule is not enforced due to the overhead. However, a user can easily turn on the rule in the JSF configuration file.

Since Java exploits are primarily delivered by drive-by-downloads, applets and Java Web Start applications are the greatest security security risk to most systems. With this in mind, a user can deploy JSF to only monitor applets and Java Web Start applications with both rules. This deployment option provides the strongest protection for the riskiest applications while allowing other applications to execute without a performance penalty. Many applets and Java Web Start applications do not require high performance[13], and both are relatively rare. A recent study found that less than .1% of websites on the entire Internet contain an applet[14].

Since both applets and Java Web Start applications run inside a browser, adjusting the browser plugin settings only effects both application types. The user can deploy the tool in only his browsers by configuring the runtime parameters of the Java browser plugin with the Java Control Panel [15]. The user would set the runtime parameters to the same values used in the tool options environment variable in other deployments.

[13]Many people could simply turn off the plugin to mitigate this issue, but enterprises tend to leave it on because some employees need it: it's cheaper to have a standard, enterprise wide configuration than custom configurations for the few employees that need applets.

[14]http://trends.builtwith.com/docinfo/Applet

[15]http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/jcp/jcp.html#java

Table IV
PERFORMANCE TEST RESULTS.

Program	No Tool time(ms)	Privilege Esc. Monitor time(ms)	Privilege Esc. Monitor overhead	Both Monitors time(ms)	Both Monitors overhead	No Tool Interpreted time(ms)	No Tool Interpreted overhead
avrora	6,840	6,458	-5.6%	110,348	1513%	32,323	373%
batik	1,184	1,108	-6.4%	17,279	1359%	6,860	479%
eclipse	13,358	13,481	0.9%	505,386	3683%	188,801	1313%
fop	272	265	-2.4%	21,151	7682%	6,551	2310%
h2	6,168	6,250	1.3%	422,526	6750%	133,134	2058%
jython	1,581	1,648	4.3%	297,056	18694%	88,269	5485%
luindex	748	674	-9.9%	48,182	6338%	16,476	2102%
lusearch	586	568	-3.1%	43,848	7385%	11,504	1864%
pmd	2,103	2,122	0.9%	30,376	1344%	8,548	306%
sunflow	1,935	1,880	-2.8%	203,036	10394%	50,645	2518%
tomcat	1,635	1,614	-1.3%	24,995	1429%	9,166	461%
tradebeans	8,979	8,871	-1.2%	414,509	4516%	147,192	1539%
tradesoap	5,301	5,310	0.2%	187,791	3443%	62,169	1073%
xalan	532	561	5.5%	48,176	8952%	12,513	2251%
total	51,221	50,811	-0.8%	2,374,659	4536%	774,151	1411%

C. Effectiveness at Fortifying the Sandbox

We performed an experiment to evaluate how effective our agent is at blocking exploits that disable the sandbox. In our experiment, we ran Java 7 exploits for the browser from Metasploit 4.10.0[16] 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.

We ran the ten exploits in our set under the following conditions: without the agent, with the agent but only enforcing the privilege escalation rule, and while enforcing both rules. We ran these conditions to respectively: establish that the exploits succeed against our JRE, test how effective the privilege escalation rule is without the security manager rule, and evaluate how effective the agent is in the strictest configuration currently available. Running the privilege escalation rule alone shows how effective the current tool is at stopping applet exploits with low runtime overhead. Overall, all ten of the exploits succeed against our JRE without the agent, four were stopped by the privilege escalation rule, and 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 V summarizes our results using the specific CVE's each exploit targeted.

Table V
A SUMMARY OF CVE'S WE RAN EXPLOITS FOR AND HOW EFFECTIVE THE AGENT WAS AT STOPPING THEM IN THE FOLLOWING CONDITIONS: (1) JUST THE PRIVILEGE ESCALATION RULE ENFORCED AND (2) BOTH RULES ENFORCED. BLOCKED EXPLOITS WERE STOPPED BY THE AGENT BEFORE THE MALICIOUS PAYLOAD COULD RUN, BUT FULLY EXECUTED EXPLOITS WERE ABLE TO COMPLETE THEIR MALICIOUS OPERATIONS.

Exploited CVE	**Privilege Escalation Enforced**	**Both Rules Enforced**
2011-3544	Fully Executed	Blocked
2012-0507	Blocked	Blocked
2012-4681	Fully Executed	Blocked
2012-5076	Fully Executed	Blocked
2013-0422	Blocked	Blocked
2013-0431	Blocked	Blocked
2013-1488	Fully Executed	Blocked
2013-2423	Fully Executed	Blocked
2013-2460	Blocked	Blocked
2013-2465	Fully Executed	Blocked

VII. 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, we believe our rules substantially improve Java sandbox security, and that future work will be able to build upon these results to create mitigation techniques for additional types of exploits.

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 of the code for any of the applications, 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 problems by using our FindBugs plugin and our JVMTI agent, which provided reviewers consistent data about where to look in the code and a means of validating their understanding of the code respectively. Furthermore,

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

we inspected entire source files when inspecting code that contained security manager operations, but it's impossible to achieve a perfect understanding in every case in a reasonable period of time without reaching out to the developer that wrote the code. Finally, we tested our tools and processes in a pilot study to find and mitigate sources of inconsistencies.

External Validity: The applications in the study were limited to open source programs, specifically well known applications included in the Qualitas Corpus and publicly available applications available on GitHub. It is possible that closed source applications interact with the security manager in ways that we did not see in the open source community, although we did inspect a few small applications with our aerospace collaborators and did not find any code that suggested this is the case.

Reliability: While the majority of the study is easily replicable, certain aspects of the studies results change over time. GitHub search results are constantly changing, thus using GitHub to generate a new dataset using our method would likely generate a different dataset. Furthermore, applications on GitHub can become inaccessible. Over the course of our security manager study, 2 applications either became private repositories or were removed from GitHub (FileManagerFtpHttpServer and Visor).

VIII. Conclusion

Our study of Java sandbox usage in open-source applications found that the majority of such 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 runtime. 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 people who attempt to make non-trivial use of the sandbox often do so incorrectly.

We indirectly observed many developers struggling to understand and use the security manager. 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 to be confused about how permissions work. In particular, they don't realize that restricting just one permission, but allowing all others enables a *defenseless* sandbox. In general, 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, better error messages, should be dedicated to helping developers correctly implement the sandbox.

References

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

[2] A. Singh and S. Kapoor, "Get Set Null Java Security," June 2013.

[3] D. Svoboda, "Anatomy of Java Exploits."

[4] "Security Vulnerabilities in Java SE," Technical Report SE-2012-01 Project, Security Explorations, 2012.

[5] "Recent Java exploitation trends and malware," technical report, Black Hat, 2012.

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

[7] 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 *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pp. 336–345, Dec. 2010.

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

[9] 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 *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.

[10] "Permissions in the JDK," 2014.

[11] 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.

[12] F. Besson, T. Blanc, C. Fournet, and A. Gordon, "From stack inspection to access control: A security analysis for libraries," in *17th IEEE Computer Security Foundations Workshop, 2004. Proceedings*, pp. 61–75, June 2004.

[13] E. W. F. D. S. Wallach, "Understanding Java Stack Inspection," pp. 52–63, 1998.

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

[15] C. Fournet and A. D. Gordon, "Stack Inspection: Theory and Variants," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, (New York, NY, USA), pp. 307–318, ACM, 2002.

[16] 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, 2007. SP '07*, pp. 149–163, May 2007.

[17] T. Zhao and J. Boyland, "Type annotations to improve stack-based access control," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pp. 197–210, June 2005.

[18] "Vulnerability Summary for CVE-2012-0507," June 2012.

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

[20] "Vulnerability Summary for CVE-2012-4681," Oct. 2013.

[21] 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.

[22] D. Svoboda and Y. Toda, "Anatomy of Another Java Zero-Day Exploit," Sept. 2014.

[23] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 212–223, ACM, 2010.

[24] D. Li and W. Srisa-an, "Quarantine: A Framework to Mitigate Memory Errors in JNI Applications," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, (New York, NY, USA), pp. 1–10, ACM, 2011.

[25] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the Native Beast of the JVM," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 201–211, ACM, 2010.

[26] M. Sun and G. Tan, "JVM-portable sandboxing of java's native libraries," in *Computer Security - ESORICS 2012* (S. Foresti, M. Yung, and F. Martinelli, eds.), no. 7459 in Lecture Notes in Computer Science, pp. 842–858, Springer Berlin Heidelberg, Jan. 2012.

[27] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), pp. 281–290, ACM, 2010.

[28] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and Detecting Malicious Flash Advertisements," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, (Washington, DC, USA), pp. 363–372, IEEE Computer Society, 2009.

[29] 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.

[30] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead Analysis and Detection of Malicious Java Applets," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, (New York, NY, USA), pp. 249–257, ACM, 2012.