Fortifying the Java Sandbox

Zack Coker, Michael Maass, Tianyuan Ding, and Joshua Sunshine
School of Computer Science
Carnegie Mellon Univeristy, Pittsburgh, PA
{zfc, mmass}@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 of X%. Running both monitors stops all ten exploits with significant overhead of XXX%, which may be acceptable when running applets since they represent the biggest danger.**

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 exploits. 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.

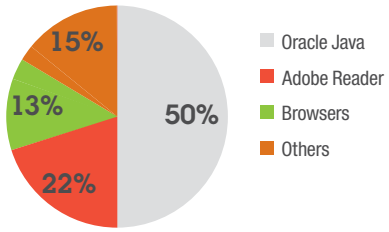

Figure 1. Most targeted applications in the enterprise, according to a Dec. 2013 survey of Trusteer customers [6].

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 III).
- An analysis of privilege escalation in the Java security model and recent Java exploits (Section IV).
- 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

A. The Java Sandbox

The Java sandbox protects an application by assigning permissions to individual classes and then enforcing the permissions through permissions checks. 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]). The application defines how to assign classes to different protection domains, as well as the specific permission set for each protection domain, based on the permissions granted in the policy. The policy specifies the permissible behavior for the application. The sandbox restricts the behavior of the application to what is allowed in the policy. By default, applications which are executed from the local file system are run without a sandbox. Web applets, on the other hand, are set to run inside a sandbox by default, preventing the applet from performing malicious operations to the determent of the host system.

Even if a policy is defined, the policy will not be enforced unless the sandbox is activated. The sandbox is activated by setting the security manager for the system,. This security manager acts as the gateway between the sandbox

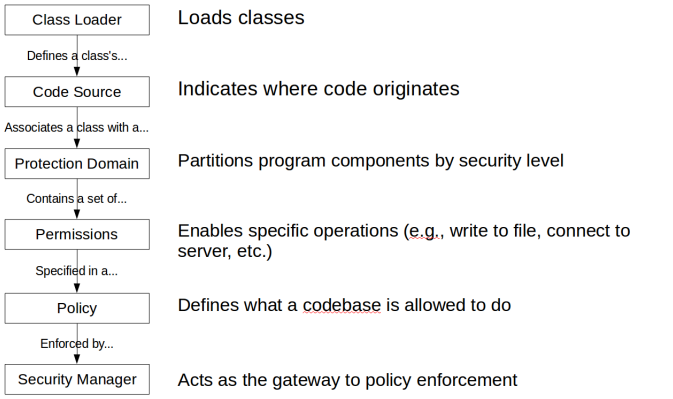

Figure 2. A high-level summary of the Java sandbox as it pertains to this work.

and the rest of the application. Whenever a class attempts to execute a method with security implications inside a sandbox, the security manager handles the permission check. For example, if an application attempts to write to a file (e.g `java.io.FileOutputStream`) inside a sandbox, the sandbox will ensure that this location inside the application has permission to write a file. The permission check is usually verified by the security manager checking each class in the current stack frame to make sure each class has the required permission. The permission check usually checks all of the classes in the stack frame, going from the class that attempted the write to the main class of the application. However, the stack walk can be stopped by a privileged class, when the privileged class specifically wraps the executing section 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 Network Protocol, a self-protecting security manager is set before the applet is loaded from the network. The security manager, and thus the sandbox, is self-protecting in the sense that the sandbox does not allow the application to change the settings of the sandbox during execution. A security manager can also be defenseless, meaning not self protecting. A defenseless manager does little to improve the security of the Java application being sandboxed. However, in our study, we have come to find that defenseless security managers have uses in certain applications III-C. Table I summarizes the set of permissions used to distinguish between self-protecting and defenseless security managers. A security manager enforcing a policy that contains even one of the listed permissions is

```
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.

defenseless. A subset of the permissions in this list were identified in [4].

B. Java Exploits

Malicious drive-by downloads using Java applets as the vector were widely reported between 2011 and 2013 (&& probably should add a citation here &&: http://java-0day.com/). While Java applets should prevent malicious applications from executing their payload, 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 Java sandbox, allowing previously constrained classes to perform any operation that the JRE can perform, meaning the malicious application can now execute the payload on the host system. 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]).

A 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 exploit, the exploit convinces a privileged class to return a reference to a class which performs privileged operations without security checks, such as the classes in the sun package. These privileged classes without security checks are isolated from a self-protecting sandbox, and only callable by classes which have already performed security checks. However, when untrusted code is able to gain direct access to privileged classes without security checks, actions with security effects can be executed in a way that avoids the restrictions of the sandbox. Once an exploit gains access to a privileged class without security checks, the privileged class is usually used to remove the sandbox (see CVE-2012-4681 [20]), allowing the exploit to execute its payload.

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

III. 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. Our empirical analysis aimed to validate the following claims, roughly categorized by the strength of the mitigation that is possible if the claim is true:

Weak Claim: *Benign applications do not disable the security manager.* If this claim 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`.

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

Moderate Claim: *Benign applications do not weaken the security manager*. Validation of this claim 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 Claim: *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-A) is disallowed. To implement this claim 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 claim, this mitigation is significantly easier to implement and is therefore stronger.

Ideal Claim: *Benign applications do not change a set security manager.* If the study supports this claim, 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 III-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)[1]. 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.

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

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 claim.

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.

B. The Security Manager Dataset

The Qualitas Corpus is a curated collection of open source Java applications for use in reproducible software studies. Table II contains a list of the 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[2] 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 GitHub. Table III contains the 17 applications we included from GitHub.

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 IV 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.

```
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.

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 JNLP 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 claims 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 claims. Due to their effect on our claims, 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[3] 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 claims 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.

1) Evaluation of our Claims: We only require one counterexample to falsify a claim from section III. This section

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

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

Table II
STUDIED APPLICATIONS FROM THE QUALITAS CORPUS

Application Name	Version Studied	Description
(Apache) Ant	1.9.4	Java Project Builder
(Apache) Batik	1.7	SVG Image Toolkit
C-JDBC	2.0.2	Database Clustering Middleware
Compiere	3.3	Business Management Tools
(Apache) Derby	10.10.2.0	Relational Database
DrJava	20130901-r5756	Lightweight Development Environment
Eclipse	4.4	Integrated Development Environment
FreeMind	0.9.0	Mind-Mapping Tool
Galleon	2.5.5	Media Server
(Apache) Hadoop	2.4.1	Distributed Computing Framework
Hibernate	4.2.2	Object-Relational Mapping Tool
HyperSQL	2.3.2	SQL Relational Database
JBoss	5.1.0.GA	Application Middleware
JRuby	1.7.13	Ruby Interpreter
(Apache) Lucene	4.9.0	Search Software
(Apache) MyFaces	2.2.4	Server Software
NekoHTML	1.9.21	HTML Parser
Netbeans	8.0	Integrated Development Environment
OpenJMS	0.7.7-beta	Messaging Service
Quartz	2.2.1	Job Scheduler
QuickServer	2.0.0	TCP Server Framework
Spring Framework	4.0.6	Web Development Library
(Apache) Struts	2.3.16.3 GA	Web Development Library
(Apache) Tapestry	5.3.7	Web Development Library
(Apache) Tomcat	8.0.9	Web Server
Vuze	5.3	File Sharing Application
Weka	3.6	Machine Learning Algorithms
(Apache) Xalan	2.7.1	XML Transforming Library
(Apache) Xerces	2.11.0.0	XML Parsing Library

Table III
STUDIED APPLICATIONS FROM GITHUB

Application Name	Git Commit Studied	Description
AspectJ	d0b8c7a1bfbc2b2f92b22bcf63598ab2442781b6	Java Extension
DemoPermissions	907dfc7610da3b0e1df76ca6b561cfbc4c60f158	Spring Extension
driveddoc	12993baabfd0dd0ca629e4bb8046097f290d1bb8	Application Connector
FileManagerFtpHttpServer	02f775b196ed6eae8e0cd2a7760193c315846498	FTP Server
Gjman	79c668c24ca65c33dc9d48d2b8372cea112ad59d	Development Toolkit
IntelliJ IDEA Community Edition	4ec1634e99ab375bb44ecf2b22a62ee4f0e39a4d	Integrated Development Environment
Jmin	9cec118eceb92b008f183d15cc9f991a98a88402	Lightweight JDK
MCVersion-Control	74b5e6d5c055a6fd204bac8ea3300626d70bd6cb	Minecraft Version Changer
NGOMS	35349cca1c518382d30f0267ef077a0a1bf52606	Business Management Tool
oxygen-libcore	79a44848bcbb39474864610cab59d0fc170ae722	Android Development Library
refact4j	fe0cdc5eb70c492993dfb55c39f5a90294383fa1	Generic and Functional Programming Framework
Security-Manager	96651247e313dd4662e52a6f8949632fdee2793e	Alternate Security Manager
Spring-Modules	583b9c78f663720f6a4433c488614fd8f18f82d2	Spring Extension
System Rules	baea2a647da1ab4965c9d4ad8a232786ea80ce1a	JUnit Extension
TimeLag	817075e61b8fbf02b65326e9ba4af7c118679b77	Sound Application
TracEE	c05cb9e8127a39017202e5bfa213d1879e6bdbc7	JavaEE Support Tool
Visor	31e032ac14d0d423e1b585de7041c054ddf83b0e	Closure Library

Table IV
CLASSIFICATION OF APPLICATION INTERACTIONS WITH THE SECURITY MANAGER

Type of Interaction	Qualitas Dataset	GitHub Dataset	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

summarizes how our claims held up against the results of this study.

Weak claim: *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 III-C2. However, some of these applications turned off the sandbox temporarily to update the imposed security policy.

Moderate claim: *Benign applications do not weaken the security manager.* This claim 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 claim: *Benign applications do not change the security manager if a self-protecting security manager has been set*. This claim 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.

Ideal claim: *Benign applications do not change a set security manager.* This claim 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. These applications tended to use the sandbox to enforce architectural constraints when interacting with other applications or forcibly disable 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.

```
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.

GJMan also enforces this architectural constraint, as shown in figure 6. The code references a blog post[4] 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

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

```
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.

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: Java automatically sandboxes applets and applications launched via Java Web Start[5]. These applications typically need to be written to comply with the default security policy that Java imposes on applications from remote sources. In cases where this is not possible, Java allows applications to run without a sandbox after obtaining user approval. Some developers cause their applications to throw a `SecurityException` when the sandbox is enabled to force users to turn the sandbox off if they want to use the application. We found two applications that were using 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 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 this method. 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

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

```
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.

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 applications during a build under a user specified set of permissions. To provide this capability, Ant sets the security manager before executing the Java application and then removes the manager after the application has finished executing. Netbeans performs a similar operation. In both of these cases a defenseless security manager is required, but a better approach would use a custom class loader to load the untrusted classes into a constrained protection domain.

Freemind 0.9.0 tried to solve a similar problem and ended up illustrating the dangers present when using a defenseless manager. Freemind is a mind mapping tool that allows users to execute Groovy scripts embedded in a map they have open. Freemind turns on the sandbox before executing groovy scripts to prevent scripts from executing malicious operations and turns the sandbox off immediately after the groovy scripts finish. The policy Freemind enforces prevents Groovy scripts from creating network sockets, accessing the filesystem, and executing programs on the machine. In the version we analyzed, these goals were not achieved. By enforcing this narrow policy and allowing the manager to be set and unset at will, Freemind accidentally grants scripts permissions that enable them to disable the sandbox. We crafted a mind map to exploit this vulnerability to execute arbitrary code and reported the issue to the Freemind developers. We included advice about how to achieve the desired outcome securely with our report.

WildFlySecurityManager is 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`.

IV. Privilege Escalation in the JVM

Section II-B provided background on exploits that attack privilege escalation vulnerabilities in Java code. Essentially, these exploits either (1) exploit a privileged class loader to load a payload class with all permissions or (2) attack a confused deputy they have direct access to. The latter 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[6].

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 can leverage these distinctions to further fortify the sandbox.

V. Rules for Fortifying the Sandbox

Given the results of our investigation in section III and the discussion in section IV, 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 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 III we discussed (1) four claims 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 claim 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[7]. 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

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

[7]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.**

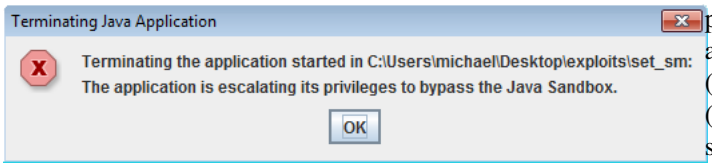

Figure 9. A popup from our agent after it caught an exploit breaking the privilege escalation rule.

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.

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 9 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.

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[8] 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[9] 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

[8] `Class.getProtectionDomain()`
[9] `RuntimePermission("getProtectionDomain")`

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 III-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 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 using the converge switch and (info about the settings used to run benchmarks). We ran the benchmarks on an otherwise idle server running Red Hat 4.1.2-12 with 32 gigabytes of ram and a 64-bit quad-core Intel X5460 CPU at 3.16 GHz. We used the 64-bit version of Java version 1.7.0 update 51.

The DaCapo benchmarks do not set a security manager: 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. Modifying the rules to apply in situations where they normally wouldn't creates problems for Eclipse, Tradebeans, and Tradesoap because they fall afoul of the modified implementation of the privilege escalation rule where they don't with the unmodified version. These three applications do not appear in our results.

Our results indicate that the privilege escalation rule causes minimal slowdown with an average of X% overhead. On the other hand, the Security Manager rule produced a significant slowdown, causing applications to run over XXX times slower on average. We investigated the cause of this slowdown and found two issues: A) read and write monitors on fields in JVMTI cause the program to run without the benefit of the JIT and B) JVMTI checks if a watched object was changed every time that object is referenced. We measured how long the benchmarks took to complete in interpreted mode to measure how much of this extensive slowdown is the result of the JIT being off and how much is the result of enforcing the security manage rule. The tests determined that the interpreted applications runtime increased by 22.5 times relative to the JITed version without JSF attached. While we do not break the figures down, the rest of the slowdown is the result of issue B above, the fact that the JRE picks less effecient code paths when certain JVMTI capabilities are enabled, and the actual enforcement of our 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 of the 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

Table V
PERFORMANCE TEST RESULTS.

	Runtime (s) / Runtime relative to no tool (%)				
	1. No Tool	2. Privilege Escalation Rule	3. Both Rules	4. No Tool and Interpreted	5. Both Rules and Interpreted
Avrora	6.80 / 100	6.86 / 100.9	109.89 / 1615.28	57.04 / 838.49	103.36 / 1519.23
Batik	2.70 / 100	1.90 / 70.4	20.28 / 749.9	21.26 / 786.44	20022.8 / 1472.42
Fop	1.379 / 100	1.87 / 135.3	21.36 / 1548.06	17.31 / 1254.25	20.32 / 1472.42
H2	7.20 / 100	6.42 / 89.3	420.29 / 5840.5	258.07 / 3586.2	408.69 / 5679.3
Jython	6.00 /100	3.94 / 65.9	309.45 / 5178.54	248.258 / 4154.53	301.29 / 5042.05
Luindex	1.28 / 100	942 / 73.84	48.35 / 3790.1	43.49 / 3408.6	45.55 / 3570.3
Lusearch	1.39 / 100	1.06 / 76.40	50.24 / 3608.61	27.32 / 1962.5	37.75 / 2711.7
Pmd	3.01 / 100	2.60 / 86.45	43.23 / 1437.78	21.98 / 731.0	28.15 / 936.1
Sunflow	2.04 / 100	2.02 / 97.78	249.20 / 12163.6	114.8 / 5603.9	174.32 / 8508.8
Tomcat	2.16 / 100	2.56 / 118.20	34.49 / 1593.31	20.90 / 965.34	23.54 / 1087.4
Xalan	1.54 / 100	1.50 / 97.98	58.58 / 3816.50	29.30 / 1908.53	41.09 / 2676.93
Average		92.12	3758.40	2251.69	3085.88

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 large overhead. However, a user can easily turn on the rule in the JSF configuration file.

Another deployment option exists that makes it possible for JSF to only monitor applets and Java Web Start applications. The Java Control Panel allows users to specify runtime paramters for the Java plugin used by browsers[10]. In this deployment, a user does not set the tool options environment variable and instead sets the runtime parameters to what they would have set the variable to. This avenue can enable most people to use JSF while enforcing both rules because drive-by-downloads, not desktop applications, are the primary means of deliverying Java exploits, many applications run from the Java plugin do not need high performance[11], and less than .1% of websites on the entire Internet contain an applet[12].

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[13] 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 VI summarizes our results using the specific CVE's each exploit targeted.

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,

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

[11]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.

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

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

Table VI
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

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, 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

The main findings of the study are:

1) The majority of applications that use the sandbox do not change the SecurityManager (19/29 in Qualitas and 9/17 in GitHub). These applications either set a SecurityManager and never change it or never set a SecurityManager but are designed to work inside a sandbox if the application is ran inside a sandbox.
2) A small portion of the applications studied used the SecurityManager for non security purposes (1/29 in Qualitas and 3/17 in GitHub).
3) Multiple developers had difficult implementing the security manager correctly, as shown by the vulnerable Freemind implementation and multiple developers' comments.

From the results of this study, we

1) Determined two rules which could be used to strengthen the sandbox in a majority of applications: the Privilege Escalation Rule and the SecurityManager Rule.
2) Tested the two rules against 10 of the most popular past Java exploits and were able to stop 40% of the exploits with the Privilege Escalation Rule and 100% with the SecurityManager rule.
3) Found the Privilege Escalation Rule could be implemented with low overhead.

With this study, we were able to take the first steps to understanding how Java applications use the sandbox. While these results are only from the studied open source applications, we believe that the results will generalize to other Java applications. We also believe that the study has found many important implications for future work to build upon:

1) Extra security can be gained by restricting Java applications from using rarely used features.
2) Java applications need a way to enforce architectural constraints when running other Java applications in a way that doesn't conflict with security- such as the ability to prevent the called application from calling System.exit() without setting the sandbox.
3) There is a need to help developers correctly implement the Java 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.