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

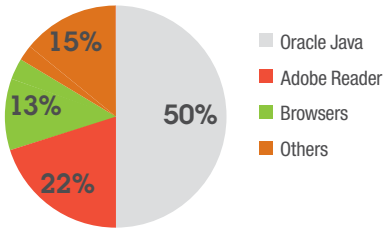

Figure 1. Most targeted applications in the enterprise, according to a Dec. 2013 survey of Trusteer customers [1].

***Abstract*—The ubiquitously-deployed Java Runtime Enviornent (JRE) routinely processes untrusted external content. The Java sandbox is designed to protect the host machine from malicious content. 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 applications and compared their use of the Java sandbox to the usage present in recent Java exploits. We found that well-secured benign applications do not modify the sandbox's security manager, the security policy enforcement mechanism, after it is first set and do not attempt to use priviledged 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 deployed and passively processes external content in the form of applets. These facts, combined with the hundreds of recently discovered vulnerabilities in Java, including a zero-day vulnerability (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.

A security goal of Java, is to allow applications to safely execute untrusted code in a sandbox so that the application and the host machine are protected from malicious behavior. However, the exploits cited cited above show that this goal has not been achieved. Past investigations of Java security exploits have shown Java malware commonly alters the sandbox's settings [2]. Typically, the exploit disables the security manager, the component of the sandbox responsible for enforcing the security policy [3], [4], [5], [6].

This knowledge leads to the question: can malicious applications be uniquely detected and stopped at runtime based on how they interact with the sandbox? To answer this question, it is important to know how benign applications interact with the sandbox, to avoid falsely halting an application that the user wants to run. After a thorough literature review, the authors were unable to find information on how Java applications interact with the sandbox.

We set up an investigation of 29 applications from the Qualitas Corpus and 17 applications from GitHub to gain an understanding how the applications interacted with the sandbox. Specifically, the investigation focused on how the applications interacted with the SecurityManager since the SecurityManager is the developer facing side of the sandbox. Applications were first analyzed statically with 2 tools: one that located possible initializations of the sandbox and another that found possible sandbox interactions. The applications were then all manually inspected to analyze the output of the static analysis tools in context and to understand important sandbox interactions. Finally, application runs were monitored to see how the sandbox changed at runtime, verifying the results of the static analysis.

Based on the findings of this analysis, two rules for fortifying the sandbox were implemented: the Privilege Escalation rule, which prevents applications from loading a class with less restrictions from a restricted class when a sandbox is set, and the SecurityManager rule, which prevents changes to the sandbox when a self-protecting Sandbox is set. These fortifications were found to allow benign applications to execute while stopping a significant portion of malicious exploits.

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

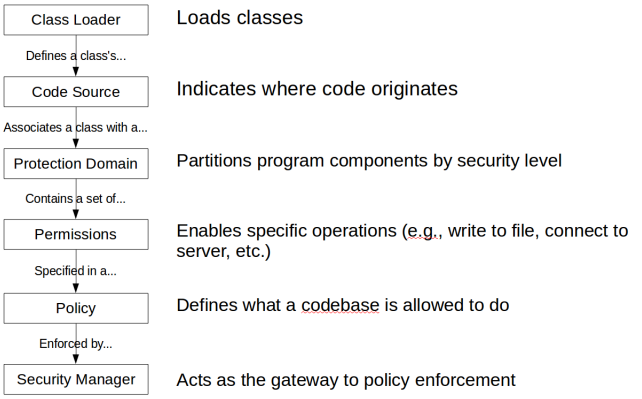

Figure 2. A high-level summary of the Java sandbox as it pertains to this work.

```
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.

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 [7]). 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 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 [8], [9], [10], [11], [12], [13], [14].

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 defenseless. A subset of the permissions in this list were identified in [5].

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

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

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 [15]).

A prominent subclass of Java exploits take advantage of a confused deputy vulnerability [16], 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 [17]), 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" [18]. For example, Svoboda [4], [19] 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`.

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) [20] 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 [21] 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.
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 to where the security manager was 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. 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.

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

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	Integerated 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

```
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.

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.

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 additional 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 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: When investigating the applications in both of the datasets, the investigators discovered applications used the sandbox in ways that were not designed to increase the security of the system. Specifically, the sandbox was used to enforce architectural constraints when interacting with other applications and to reduce development complexity in web application development.

Enforcing Architectural Constraints : Java applications commonly call `System.exit()` if the application throws a non-recoverable error condition. However, this error handling method causes problems when applications work together, specifically when an application calls another ap-

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

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

plication which will exit on an error. The problem with this interaction is that when the called application executes the `System.exit()` line, the called application kills the calling application as well. The calling application is killed along with the called application because the execution of `System.exit()` stops the virtual machine instance in which both applications are running.

In many cases, the ability for the called application to kill the calling application is an unintended side-effect. Thus the calling application needs to enforce the architectural constraint that called applications can not terminate the execution of the calling application. To enforce this architectural constraint, Java applications that call other applications set a SecurityManager that prevents `System.exit()` calls. The new SecurityManager usually stops all calls to `System.exit()` while the new SecurityManager is set and, if a SecurityManager was previously set, the new SecurityManager defers all other security checks to the previously set SecurityManager. When the calling application determines that the called application has finished, the calling application usually restores the previously set SecurityManager if one exists.

One example of an application preventing another application from calling `System.exit()` is Eclipse in Qualitas which calls Ant. When Ant reaches an unrecoverable error condition, Ant will call `System.exit()` to terminate the compilation. However, Eclipse wants to continue executing and report an error to the user if Ant runs into a error condition.

```
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      }
```

Shown in the code above, on line 691 Eclipse sets a SecurityManager to prevent Ant from calling `System.exit()`. After performing some other checks, Ant is executed. Then after handling other error conditions, the original SecurityManager is restored.

Another example of enforcing the architectural constraint occurs in GJMan in the GitHub data set. The code references a blog page describing this problem and the implemented solution: http://www.jroller.com/ethdsy/entry/disabling_system_exit. The code reads

```
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  }
```

The code contains the allow method which creates a SecurityManager that stops System.exit calls and the sets the created SecurityManager as the Sandbox for the Java Virtual Machine. The file also includes a method to remove the SecurityManager and removes the Sandbox from the Java Virtual Machine. While GJMan does not execute these lines explicitly, GJMan is written to be a library for other applications, so this file is likely used in other applications.

In total, we found 3 applications using a variation of this technique: Eclipse, GJMan, and AspectJ. While this technique is useful in applying architectural constraints, and probably the best architectural solution available in Java at the moment, this technique is likely to cause problems when applications desire the sandbox to also enforce security constraints. The reason this technique creates problems when trying to enforce security constraints is that the sandbox must be set in a state which it can be dynamically removed, otherwise the calling application could never terminate. By allowing the sandbox to be dynamically removed, the application must be carefully written to avoid allowing malicious code to turn off the sandbox.

Reducing Web Application Development Complexity: When Java web applications are run inside modern web browser, the application is sandboxed to protect the machine which hosts the browser. When the sandbox is set up, applications have to work inside the restrictions of the sandbox, meaning that applications are only approved to use the permissions allowed in the standard web browser sandbox.

To support applications which need to use permissions unavailable in the standard web browser sandbox, web browsers allow Java web applications to run without a sandbox after obtaining user approval. Developers, aware that users can turn off the sandbox, can develop applications in a way that require the sandbox to be explicitly turned off. Specifically, these applications ensure that the sandbox is off at the start of executing the web application. If a default sandbox is set, this check will cause the applications to crash with a SecurityException. Thus, the only way to run the applications are to turn off the sandbox.

In total we found two applications that were using this method: Eclipse in Qualitas and Timelag in Github. Since malicious applications are also known to turn off the sandbox, i.e. null the SecurityManager, it is important to carefully distinguish between a benign application which turns off the sandbox and a malicious application which turns off the sandbox. In the case where the application attempts to turn off the sandbox when starting the application and the application

does not attempt to ensure the turn off attempt will succeed, it is determined that the application is trying to check if a sandbox is set and, if so, stop the application. On the other hand, applications which attempt to ensure that the sandbox is turned off, even when one is set, are likely malicious.

Eclipse contains the following code section in the file WebStartMain.java

```
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.
```

The Eclipse comments show that the attempt to turn off the sandbox was done to avoid the permission issues caused by the default sandbox for Java Web Start. Timelag also contains the `System.setSecurityManager(null);` line as the first line of its main function in the file JTimelag.java but does not contain any comments. So the motivation behind turning off the sandbox at the beginning of execution had to be inferred.

3) Changing the SecurityManager for Security Purposes: In both datasets, we found applications that set a SecurityManager at one point a program and then either explicitly change or allow the user to change the currently set SecurityManager. Three applications allow the user to set and change the currently set SecurityManager using provided methods (Batik, Eclipse, and Spring-modules) while three applications attempted to explicitly set and change the SecurityManager during execution (Ant, Freemind, and Netbeans (!! Not 100% sure about this one from what Tianyuan wrote, will need to investigate it later!!)).

For the applications that allowed the user to set and change the SecurityManager, it seems the applications are attempting to provide the flexibility of the current Sandbox implementation in Java, with no strong security case for implementing the SecurityManager interactions this way.

For example, in Batik, the file ApplicationSecurityEnforcer contains the code segment:

This method was designed to allow users to add a SecurityManager to an application that uses the Batik library. The method first takes a Boolean, then checks to make sure a Batik specific SecurityManager is not set. If a SecurityManager defined outside of Batik was not set, the application either sets a Batik SecurityManager, which refreshes the policy for the SecurityManager or sets the SecurityManager to null which turns off the sandbox. This methods use was demonstrated in some of the examples provided in the application download from the Batik website. Two of the downloads provide ways to set a SecurityManager at start up, the squiggle browser demo and the rasterizer demo. While the squiggle browser demo sets a SecurityManager and never changes it, the rasterizer

```
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         ...
```

demo can be set to call enforceSecurity with a true argument the first time and a false argument the second time, cause the application to set a SecurityManager and then remove it at the end of the demo. While this was an interesting occurrence, there seemed to be no valid reason, other than showing off the capabilities for the library, to turn the SecurityManager off at the end of the execution.

Only in the case where an application can not be reset and the application cannot predict future required permissions, which we believe to be rare, applications would need the ability to dynamically refresh the sandbox. In all other cases, the ability to dynamically adjust the SecurityManager and the sandbox's policy is not required. As long as the required permission can be predicted before the application is running, class loaders with defined permission sets can assign the permission to any newly loaded Java classes, thus allowing applications to provide permissions to the classes that require them. (&& I don't like this section's wording - I should ask Michael for input on how to word it correctly).

As mentioned earlier, Ant, Freemind, and Netbeans explicitly set and then change the current SecurityManager during runtime. Ant wants to allow the user the capability to execute Java applications during a build under a user specified set of permissions. To provide this capability, Ant sets the SecurityManager before executing the Java application and then removes the SecurityManager after the application has finished executing. Netbeans also takes a similar approach (&& need to check this &&).

Freemind 0.9.0 also tried to solve a similar problem, but demonstrates the difficulty of correctly implementing this solution. Freemind is a diagram drawing tool that allows users to execute Groovy scripts on the current drawing. The developers of Freemind implemented the sandbox so that it would turn on before executing a user run script and would turn off after the script finished executing. The security goals of the Freemind sandbox was to stop malicious scripts from creating files, executing programs on the machine, and creating network sockets to establish connections with outside entities. Unfortunately, in the version we analyzed , these goals

were not achieved. By implementing the SecurityManager in the old SecurityManager style, explicitly removing privileges, multiple dangerous permissions were left, such as the ability to alter private variables with reflection. This privilege made it trivial to remove the currently set SecurityManager inside a Groovy script, thus turning off the sandbox, and allowing the script to create files. The authors submitted a sample exploit to the Freemind development group and made recommendations on how to improve the security of the Freemind sandbox. (*** also should probably mention something about the SecurityManager indirection ***).

As with the applications that allow setting and chaining the SecurityManager, we believe the applications which set and change the SecurityManager explicitly can execute correctly using a static SecurityManager with class loaders which limit the permissions of the restricted code sections.

The one application that did not fall into either of these categories was WildflySecurityManager. The WildflySecurityManager allows turning off the permission checks for classes which have been granted the DO_UNCHECKED_PERMISSION. However, this method of permission checks seems to be the same as assigning the privileged classes the AllPermissions and then executing doPrivileged on the privileged actions that the privileged class needs to do.

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 former is quite rare and typically the fault of a vulnerable third-party library because directly exposing privileged classes is a violation of the *access control* principle that is part of the Java development culture[4].

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 more privileged than itself 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 because classes should not be loading more privileged classes in the presence of a self-protecting security manager because a self-protecting security manager will not allow the creation of a privileged classloader to load a class with all permissions (see the createClassLoader permission in table I).

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

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 load a more permissive 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. This rule is based around the fact that many exploits load a class with more privileges than the calling class to break out of the sandbox.

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. Thus the rule has to make an exception for these classes. Specifically, the rule makes exceptions for classes returned by the call `java.security.Security.getProperty("package.access`

B. Security Manager Rule

The Security Manager Rule states that a Security Manager cannot change if a *self-protecting* Security Manager has been set by the application. By setting a *self-protecting* Security Manager, the application is removing ability of changing or removing the sandbox. Thus, the rule is violated if something in the application causes a change in the sandbox's setting, which is what many exploits try to ensure will happen. (&& probably should tie these to the hypotheses somehow but need to read the paper to understand how to do that.&&)

VI. Mitigations

In section III we discussed (1) three claims that would lead to Java exploit mitigations if validated and (2) how we went about validating them. In section III-C we discussed additional research questions we answered 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 that can be enforced to stop current exploits. In this section we discuss the implementation and evaluation of a tool that implements the privilege escalation and SecurityManager rules. We evaluated

this tool 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 [22], [23], [24], [25], and the machine learning community has put some effort into detecting exploits delivered via drive-by-downloads using Java applets and similar technologies [26], [27], [28], [29]. We implemented a tool in JVMTI to proactively stop exploits written directly in the Java programming language to exploit vulnerable Java code[5]. In particular, our tool focuses on fortifying the Java sandbox by reducing the methods that the sandbox can be successfully bypassed in an attack.

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 modifications, 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 SecurityManager 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 SecurityManager 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 [24] 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 [22], 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.

[5]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.

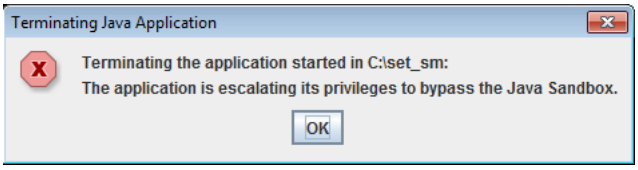

Figure 5. A popup from our agent after it caught an exploit breaking the privilege escalation rule.

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. In monitor mode the agent logs the offending behavior but leaves the JVM's execution of the application untouched. In either case, 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 5 shows an example of a popup displayed after an exploit was caught breaking the privilege escalation rule.

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.

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[6] 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[7] 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 `AllPermissions` 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 `AllPermissions` 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 in 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 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

Overhead is an important metric of security tools. Specifically for these rule implementations, the important metrics which need to be investigated are performance overhead, memory overhead, and user overhead.

Performance overhead was tested using version 9.12 of the Dacapo Benchmark [30], a standard set of Java applications with non-trivial memory loads. The applications in the benchmark are Avrora, Batik, Eclipse, Fop, H2, Jython, Luindex, Lusearch, PMD, Sunflow, Tomcat, TradeBeans, TradeSoap, and Xalan. Performance was measured by running the provided section of the application 5 times and the the mean of the runtimes was computed. All performance tests were ran on an 8 Intel i7 core HP Envy M4 laptop with 8 GBs of RAM.

Since the Dacapo Benchmarks did not set a security manager during any of the benchmark runs, the the tool had to be altered to work in all cases, instead of when a security manager is set. (add something about how the altered version produces the maximum overhead possible) This created

[6] `Class.getProtectionDomain()`

[7] `RuntimePermission("getProtectionDomain")`

problems for Eclipse, Tradebeans, and Tradesoap where the altered Privilege Escalation Rule implementation prevented the benchmark from running. Catching the applications with the altered Privilege Escalation Rule implementation is not a problem because the non-altered Privilege Escalation Rule implementation can successful execute the Eclipse, Tradebeans, and Tradesoap benchmarks.

The results of the performance tests indicate that the Privilege Escalation Rule implementation caused minimal slowdown, and actually ran 8% faster than the benchmarks without a tool set. This increase is believed to be due to the fluctuations between runs and not due to any effects of the tool. The Security Manager Rule implementation, on the other hand, produced a significant slowdown, causing applications to run over 37 times slower on average. We investigated the cause of this slowdown and found two causes 1) checking for variable changes in JVMTI cause the program to run in interpreted mode and 2) JVMTI checks if the watched object reference was changed at every object reference. To determine how much of a speed increase could be gained by implementing the Security Manager Rule using a tool that does not require the application to run in interpreted mode, we measured how long the benchmarks took to complete in interpreted mode. The tests determined that the interpreted applications runtime increased by 22.5 times the original Just-In-Time Compiled runtime. While this is not the full portion of the slowdown for the Security Manager Rule implementation, running the application in interpreted mode is over 60% of the slowdown. It is likely that the Security Manager implementation could also be improved by avoiding the overhead associated with checking for a security manager change in every object change. Finally, to confirm that the Security Manager Rule implementation was running in interpreted mode, the benchmark was timed with both tools in interpreted mode. While running the tool in interpreted mode was faster than running the tool without interpreted mode explicitly set, the author believe this result is either due to the variations of individual performance tests or that explicitly setting interpreted mode causes the mode switch to occur faster.

Future work could address the slowdown associated with the Security Manager rule implementation. These approaches could focus on the underlying reasons for the slowdown in the tool or re-implement the rule in a different way.

The authors hypotheses that the Security Manager Rule can be implemented with significantly less performance overhead if the rule is built into the JVM. The problem with implementing the rule by changing the JVM is that any unofficial changes would not be supported by the JVM developers, thus increases the difficulty to update the JVM to the latest released version. Thus, the rule would likely need to be implemented by the JVM maintainers to create a significantly faster version without losing the ability to update the JVM.

The memory overhead of the tool is likely very small relative to a large program. The tool only copies a couple of memory references, such as the loaded classes permission set, when performing both types of security checks. The copies are also deallocated after each check. A few objects, such as the system's security manager and references to repeatedly called methods in the JVM are saved between security checks. All objects created by the tool likely have a minimal memory footprint compared to the complete application, although the memory footprint was not explicitly measured in this study.

The user overhead of the tool is also minimal. 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 the tool is compiled, the tool 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 point to the library file for the tool with the command -*agentpath=path_to_library* 2) set the JSF_HOME environment variable to point to the directory with the appropriately named configuration file. JAVA_TOOL_OPTIONS is the environment variable provided by Oracle that allows executing a JVMTI agent for all instances of the JVM. The JSF_HOME environment variable is an environment variable only applicable to the tool. These two options can be set for the whole system or only a terminal using a short script.

The default configuration of the tool is the Privilege Escalation Rule implementation is turned on while the Security Manager Rule is turned off. The default mode of the tool is enforce mode, which was described in the JVMTI Implementation section. The other default configuration option is to output the security logs to a file named jsf.log in the current directory of the running application. All of these options can be changed in the configuration file. The default configuration of only the Privilege Escalation rule implementation provides the security benefit of the Privilege Escalation Rule without incurring the performance hit of the Security Manage Rule implementation, which is discussed in the next section. The Security Manager Rule implementation can be turned on in the configuration file, adding an extra check while incurring the performance cost of the Security Manager Rule implementation. The tool can be switched from enforce mode to monitor mode if only the log files are desired. The resulting logs can also be set to a single file by specifying the file's path in the configuration file.

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[8] 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

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

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

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

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.

D. Limitations

Neither of these rules will able to stop 100% of all Java exploits. Both rules are unable to catch exploits which are able to escape the sandbox without violating the constraints the rules impose. While the privilege escalation rule is able to stop many of the most common Java exploits (40% of tested exploits), the rule still does not catch a significant portion of the exploits. The SecurityManager rule is also not able to catch all exploits. While the SecurityManager rule was able to catch all tested exploits, Java makes it possible to write exploits that do not turn off the SecurityManager but are still able to cause significant damage. While the authors believe the rules created in this study provide a substantial improvement over the current sandbox implementation, the authors also believe that future work will be able to build upon the results of this study to create improved mitigation techniques.

Internal Validity: The source of data for this study was source code and comments. In most cases, security manager interactions could clearly be determined although the reason behind complex interactions may be misdiagnosed. Also comments and source code could have been taken out of context, leading to incorrect conclusions. Finally using two different reviewers leads to the possibility of different interpretations for the same data. Misdiagnosing complex interactions was reduced by using the FindBugs tool and the JVMTI tool to help with the analysis, providing a way for the researchers to test their analyses. The issue of interpreting comments and source code out of context was reduced by examining the whole file but can not be completely reduced without extra data sources, such as developer interviews. Reducing the possibility of misinterpreting these sections of the program is left to future work. The issue of using two different researchers reaching different conclusions was reduced by comparing the 6 different independent analyses when testing the methodology of the study.

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 differently, although investigating this problem is left to future work. Another possible problem for the study is that 11 out of 46 applications where created by the Apache Software

Foundation. This problem was reduced by filtering a larger set of applications initially for the Qualitas Corpus, showing that in a diverse set of 112 Java applications, 83 were determined to not interact with the security manager at all from the initial filtering process.

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 and so using GitHub to generate a new dataset using our method would likely generate a different dataset. Another problem is that applications on GitHub can become inaccessible. Ever the course of the study, 2 applications either became private repositories or were removed from Github (FileManagerFtpHttpServer and Visor). It is also possible that different researchers reach different conclusions for what the project developers intended. This threat to validity was reduced by the two investigators cross checking how 6 applications were interpreted. We expect that with similar studies will reach similar conclusions.

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

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

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

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

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.