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

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

ABSTRACT

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

1. INTRODUCTION

Java applications are very popular targets for malicious attackers (see Figure 1), for three broad reasons. First, it is the Java Runtime Environment (JRE) is widely installed on user endpoints. Second, the JRE can (and often does) execute external code, in the form of applets and Java Web Start (JWS) [2] applications [3, 4]. Finally, Java contains hundreds of vulnerabilities ***Todo 1-1:*** *Can we add raw numbers here, say over 2011-2013?*, including zero-day vulnerabilities (e.g. CVE-2013-0422). In the common scenario, often referred to as a "drive-by download", attackers lure users to a website that contains a hidden malicious applet. Such malicious applets exploit security vulnerabilities in the JRE to deliver malware, typically leaving the user unaware.

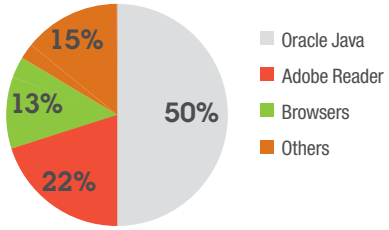

Figure 1: Pie chart showing the most targeted applications on enterprise workstations, according to to a Dec. 2013 survey of Trusteer customers [1]. Java represented half of all attack-attempts in their sample.

In theory, such attacks should not be so common: Java provides a sandboxing mechanism that enables the safe execution of untrusted code and isolates components another. Both the host application and host machine should therefore be protected from malicious behavior on the part of external code. In practice, these security mechanisms are problematically buggy, such that the sandbox often fails to properly contain external code. Java malware typically alters the sandbox's settings [5]. Such exploits take advantage of defects in either the JRE's implementation the application's configuration of the sandbox to disable the security manager, the component of the sandbox responsible for enforcing the security policy [6, 7, 8, 9].

In this paper, we investigate this disconnect between theory and practice. We hypothesize that it results primarily from unecessary complexity and flexibility in the design and engineering of Java's security mechanisms. ***Todo 1-2:*** *Can someone flesh this out? What types of complexity/flexibility does the sandbox allow that we don't need? Likely relevant example: the ability to change the security manager at runtime.* In particular, we hypothesize that benign applications interact with the security manager in ways that are measurably different from the ways that exploitative applications do. If true, this difference can be leveraged to prevent future attacks.

To validate these insights, we conducted an empirical study of benign open source Java applications. We sought to answer the research question: How do benign applications interact with the Java security manager? We identified 46 open-source Java projects that use the security manager, taken from the Qualitas Corpus [10] and GitHub. For each project, we isolated code that interacts with the security manager, manually characterized those interactions,

and constructed and used a Java Virtual Machine Tool Interface (JVMTI) agent to confirm that our characterizations were accurate at runtime.

We discovered that there are two types of security managers used in practice. *Defenseless* security managers enforce a policy that allows code inside the sandbox to modify sandbox settings. Applications with defenseless managers are inherently insecure, because externally-loaded malicious code can modify or disable the security manager. In our study, this situation typically arose in applications that modified sandbox settings at runtime, often because they use the security manager to enforce policies unrelated to security. ***Todo 1-3:*** *I think this train of thought needs one more observation, starting with something like "this is interesting because". I'm drawing a blank on what should come after the "because", however, hence this todo.* *Self-protecting* security managers do not allow sandboxed code to modify security settings. The applications in our dataset with self-protecting managers, including all applets and JWS applications, did not change those settings over the course of execution. It might still be possible to exploit such applications due to defects in the JRE code that enforces security policies, but not due to poorly-deployed local security settings.

In practice, applications that attempt to use the sandbox for its intended purpose—protection from exploitative external code—do not make use of its vast flexibility. We therefore propose two runtime rules to fortify the Java sandbox against the two most common modern attack types. We evaluate our rules with respect to their ability to guard against the ten applets in Metasploit 4.10.0[1] that successfully exploit unpatched versions of Java 7. Taken together, the monitors detected and stopped all ten exploits, and neither monitor produces false-positives for a corpus of benign JWS applications. We are engaged in an on-going discussion on the security-dev mailing list for OpenJDK about implementing runtime enforcement of these rules in the JVM itself.

The contributions of this papers are as follows: ***Todo 1-4:*** *CLG would like us to rethink these contributions once the rest of the paper has been modified.*

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

2. BACKGROUND

In this section, we describe the Java sandbox (Section 2.1), distinguish between defenseless and self-protecting Security Managers (Section 2.2) and provide a high-level description on how exploits commonly bypass these security mechanisms (Section 2.3).

2.1 The Java sandbox

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

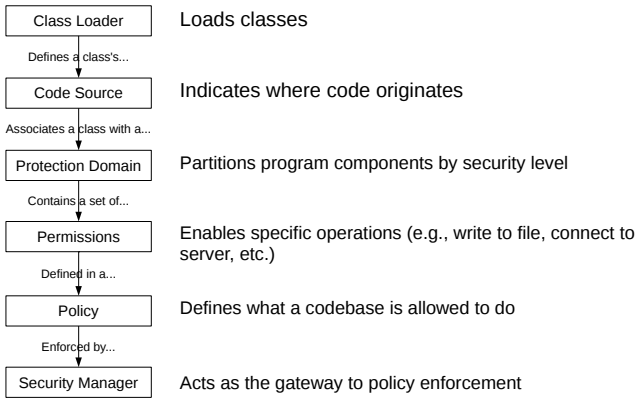

Figure 2: High-level summary of the components of the Java sandbox.

The Java sandbox is designed to safely execute code from untrusted sources. The relevant components are summarized in Figure 2. When a *class loader* loads a class (e.g., from the network, filesystem, etc.), it assigns the class a *code source* that indicates the code origin, and associates it with a *protection domain*. Protection domains segment the classes into groups by *permission set*. These sets contain permissions that explicitly allow actions with security implications, such as writing to the filesystem, accessing the network, using certain reflection features, etc (see a more complete list at [11]). Unlisted actions are disallowed for classes in that protection domain. *Policies*, written in the Java policy language [12], define the permission sets and the code sources associated with them. By default, classes loaded from the local file system are run without a sandbox; all other applications are run inside a restrictive sandbox. This prevents applications from the network or other untrusted sources from executing malicious operations on the host system.

The sandbox is activated by setting a *security manager*, which acts as the gateway between the sandbox and the rest of the application. Whenever a sandboxed class attempts to execute a method with security implications, that method queries the security manager to determine if the operation should be permitted. We refer to such methods as *privileged code*. For example, if a sandboxed application attempts to write to a file by using `java.io.FileOutputStream`, the latter will first check with the security manager to ensure that a write to that file is permitted. Missing checks in code that *should* be protected are a common source of Java vulnerabilities, because the security-critical code must initiate the check. Note that such vulnerabilities lie in the JVM itself (i.e., the code written by the Java developers), not in the code that uses the sandbox to execute untrusted external applications.

To perform a permission check, the security manager walks the call stack to ensure each class in the current stack frame has the permissions necessary to perform the action. Consider a sandboxed class `Sandboxed` that attempts to create a `java.io.FileOutputStream`. The `FileOutputStream` constructor checks with the security manager to determine whether `Sandboxed` has permission to open the file in question. The manager then checks the permission sets of each class in the call stack: `FileOutputStream`, and then `Sandboxed`. 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`. Privileged code, such as `FileOutputStream` in our example, can wrap sensitive actions in a `doPrivileged` call to preempt the stack check. This allows privileged code sections to perform actions with security implications at the request of non-privileged code.[2]

2.2 Defenseless vs. self-protecting managers

Todo 2-1: *I think we need to hammer this point a bit more: Java's security model is too complicated—meaning it's implemented badly/in a way that's full of bugs—and also unecessarily so—because people don't need all that flexibility. I'm not sure how to expand this text to help with that point-hammering, but I think it would be useful to the development of the argument. Anyone see what I'm getting at and want to try to expand accordingly?* Java is flexible about when in an application's execution the sandbox is configured and enabled. The default case for web applets and applications that use Java Web Start is to set what we call a *self-protecting* security manager before loading the application from the network. The security manager, and thus the sandbox, is self-protecting in the sense that it does not allow the application to change sandbox settings. We contrast self-protecting managers with those we call *defenseless*, meaning that sandboxed applications have privileges that permit them to modify or disable the security manager at runtime. Such a security manager is the exact opposite of self-protecting. A defenseless manager is virtually useless in terms of improve the security of either a constrained application or its host. However, we find in Section 4 that the developers of some benign applications have found interesting non-security uses for defenseless managers.

Table 1 summarizes the set of permissions used to distinguish between self-protecting and defenseless security managers. We consider any security manager that enforces a policy that contains any one of the listed permissions to be defenseless. A subset of the permissions in this list were identified in [8].

2.3 Exploiting Java Code

While the Java sandbox *should* prevent malicious applets from executing their payloads, defects in the Java Runtime Environment (JRE) enforcement of these security mechanisms permit malicious code to set a security manager to `null`. Setting the security manager to `null` disables the sandbox, allowing previously constrained classes to perform operations with the privileges of the JRE itself. This approach was taken in a large proportion of drive-by downloads exploiting Java applets between 2011 and 2013 [**?**]. There are a couple of methods by which an exploit may nullify a security manager:

Type confusion. An attacker breaks type safety to craft an object that can perform operations as if it had a different type. Commonly, attackers craft objects that either (1) point to the `System` class or (2) act as if they had the same type as a privileged class loader (see CVE-2012-0507 [20]). In the first case, any operation performed on the masqueraded class is applied to the real `System` class, allowing the attacker to directly alter the field storing the security manager. In the second case, the malicious class can load an exploitative payload with elevated privileges.

[2]Stack-based access control is discussed in more detail in [13, 14, 15, 16, 17, 18, 19].

Confused deputy. Exploitative code "convinces" another class to return a reference to a privileged class [21] known to contain a vulnerability (such as a missing security check). The attacker then takes advantage of that vulnerability to disable the sandbox (see CVE-2012-4681 [22]). The "convincing" is necessary because it is rare that a vulnerable privilege class is directly accessible to all Java applications; doing so violates the *access control* principle that is part of the Java development culture.[3] Once an exploit gains access to a vulnerable privileged class, that class can be "tricked" into executing code that disables the sandbox

The confused deputy attack is an example of privilege escalation. Most privileged classes in the JRE implement features that can be accessed via less-privileged code paths; benign applications therefore rarely need to directly access them. Because a privileged class loader must be used to load a privileged class, a non-privileged class does not typically have access to vulnerable privileged classes anyway. The danger of vulnerabilities in privileged classes is therefore mitigated, because such vulnerabilites cannot be exploited directly unless malicious code succeeds in modifying its privileges first. This redundancy is implicit in the Java security model: If any class could load more privileged classes and directly cause the execution of privileged operations, the sandbox in its current form would serve little purpose. In sections 5 and 6 we discuss how we can leverage these distinctions to further fortify the sandbox.

3. SECURITY MANAGER STUDY

Modern exploits that manipulate on the Java security manager perform one operation: They disable it. This is possible because of a complex set of vulnerabilities and misconfigurations on the part of developers of both Java applications and the JRE itself. Fundamentally, we hypothesize that these mistakes arise largely from the fact that the Java security model is extremely complex, in that it grants enormous flexibility to application developers to set, reconfigure, manipulate, weaken, strengthen, or otherwise change a security manager after it has been created.

Do applications need this power? Do they regularly take advantage of the ability to disable or weaken the sandbox? If not, we can design backwards-compatible JVM enhancements that can stop even zero-day exploits by eliminating the operations on which exploits depend. ***Todo 3-1:*** *This strategy would allow the sandbox to enforce policies on 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. CLG says: I'm not sure I understand this sentence, but I think it's important. Can we simplify it?*

In this section, we describe the methodology for and limitations of an empirical study on open-source Java projects to validate this strategy, answer our motivating question, and provide data in support of JVM enhancements to fortify the Java sandbox. We focus our efforts on the security manager, as it is the means by which applications interact with the sandbox.

3.1 Dataset

We used applications from the Qualitas Corpus (QC) [10] and GitHub to form a dataset of applications that use the

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

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

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

security manager. The Qualitas Corpus is a curated collection of open source Java applications for use in reproducible software studies. While QC provides a strong starting point for the construction of a dataset for this study, their inclusion criteria[4] means it is comprised largely of large, popular applications and frameworks. We diversified our dataset by including applications from GitHub. Table 2 lists all studied applications. Version numbers and Git commit hashes are available in an online supplement.[5]

We investigated sandbox interactions in 29 of the 112 applications from QC version 20130901. We used `grep` on the source code of the full set to find instances of the keyword *SecurityManager*, which identified 29 applications of interest. We performed a similar process on GitHub, searching only Java files. We added the keyword *System.setSecurityManager(* to remove false positives, and *System.setSecurityManager(null)* to find applications that disable the manager. We picked the top seven applications from the results for each keyword, removed false positives, resulting in an additional 17 applications that were not already in QC. We only looked at the latest commit *Todo 3-2: as of when?*.

3.2 Methodology

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

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

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

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

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

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 [23] plugin that uses a dataflow analysis to determine where `System.setSecurityManager`() is called, as well as the lines of code where its arguments were initialized. We created a dynamic analysis tool using the Java Virtual Machine Tool Interface (JVMTI) [24]. JVMTI is designed to allow tools to inspect the current state of Java applications and control their execution; it is commonly used to create Java debugging and profiling tools. Our dynamic analysis tool set a modification watch on the `security` field of Java's `System` class. This field holds the current security manager object for the application. The watch prints out the class name, source file name, and line of code where any change to the field took place. A special notice is printed when the field is set to `null`.

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

1. Run grep on all Java source files in the application

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

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

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

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

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

to output every line containing the keyword *SecurityManager* and the 5 lines before and after it.

2. Reject any application where it is clear from the grep output that the keyword is used in ways that are unrelated to the security manager class.
3. Run the static analysis on retained applications.
4. Manually inspect code identified in step 3. Start where the manager is set and trace back to locations where potential security managers are initialized.
5. Manually inspect all of the lines returned in step 1, looking for how the application interacts with the sandbox.
6. Execute the application with the dynamic analysis using parameters and actions steps 4 and 5 show affect the security manager to verify conclusions from previous steps.
7. Summarize the operations performed on the security manager with an emphasis on points that support or reject each hypothesis.

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

4. STUDY RESULTS

We characterized the security manager interactions of the applications in our dataset by assigning each one of five types. The types 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 but do not modify the manager or its policy, (4) applications that only interact with the manager in unit tests, and (5) applications that do not actually interact with the manager. Table 3 summarizes our dataset using these types.

Type 3, 4, and 5 applications will not be discussed further because their interactions with the sandbox cannot violate our hypotheses.

Type 2 applications can violate our hypotheses and therefore provide the bulk of our discussion below. However, a few Type 1 applications are discussed due to the novel insights they provide into benign interactions with the sandbox. We discuss application that use the sandbox for purposes that are not security related in Section 4.2 and applications that use the sandbox for its intended security purposes in Section 4.3.

4.1 Evaluation of the Hypotheses

We only require one counterexample to falsify a hypothesis from Section 3. This section summarizes how our hypotheses held up against the results of this study.

Hypothesis 1: *Benign applications do not disable the security manager.* The investigation determined that some benign applications do disable the security manager, but these were typically were not using the sandbox for security purposes.

Hypothesis 2: *Benign applications do not weaken the security manager.* Several applications provided methods for the user to dynamically change the security policy or the manager in ways that can reduce the security of the sandbox.

Hypothesis 3: *Benign applications do not change the security manager if a self-protecting security manager has been set.* This hypothesis was supported by both datasets.

Hypothesis 4: *Benign applications do not change a set security manager.* This hypothesis was falsified by multiple applications that changed the security manager.

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

4.2 Non-security uses of the Sandbox

This section describes applications that provide novel insights into benign sandbox interactions unrelated to satisfy security requirements. Most of these applications used the sandbox to enforce architectural constraints when interacting with other applications or forcibly disabled the sandbox to reduce development complexity.

4.2.1 Enforcing Architectural Constraints

Java applications often call `System.exit()` when a non-recoverable error occurs. This error handling strategy causes problems when applications that use it are used as libraries. When the library application executes `System.exit()`, the calling application is closed as well because both applications are running in the same JVM. This is not the desired outcome in several cases.

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 constraint by setting a security manager that prevents `System.exit()` calls.

This case appears in Eclipse, which uses Ant as a library. When an unrecoverable error condition occurs, Ant kills the JVM to terminate execution of the build script currently running. However, Eclipse should continue executing and report an error to the user when Ant terminates with an error code. Figure 3 shows how Eclipse sets a security manager to enforce this constraint right before Ant is executed. After Ant closes and any error conditions are handled, the original manager is restored.

GJMan also enforces this constraint and contains a code comment referencing a blog post that we believe is the origin of this solution.[6]

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

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

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

4.2.2 Reducing Web Application Development Complexity

We found applications that were complicated by the Java security policies for web applications (applets and applications launched via JWS). By default, Java executes such an application inside a restrictive sandbox that severely limits the operations the application can perform, excluding operations such as accessing local files, retrieving resources from any third party server, or changing the security manager.

Applications in our set that cannot run in a restrictive sandbox universally opted to run outside of the sandbox because the alternative is to painstakingly construct the application to run reasonably without required privileges (e.g. by detecting the sandbox and disabling privileged operations). To avoid executing the applet in a restrictive sandbox, a developer must get the application digitally signed by a recognized certificate authority then specify that the application should run outside of the sandbox. We found that applications using this method attempted to set the security manager to `null` at the beginning of the application, causing a restrictive sandbox to catch the security violation and terminate the application.

We found two applications that do this: Eclipse and Timelag. The rationale for disabling the manager in Eclipse is explained in a code comment that reads, "The launcher to start eclipse using webstart. To use this launcher, the client must accept to give all security permissions." Timelag performs the same operation but does not contain any comments, thus we can only infer their motivation.

4.3 Using the Security Manager for Security Purposes

This section describes applications that provide novel insights into benign sandbox interactions related to improving the security posture of the application. Several of these

```
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 4: Security manager interactions in Batik.

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

Figure 5: Example Ant build script element to grant all but one permission. This specific permission set leads to a defenseless security manager.

applications clearly violate hypotheses 1, 2, and 4: 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 4 shows an interesting case from Batik copied from `ApplicationSecurityEnforcer.java`. This method allows users to optionally constrain the execution of an an application that uses the Batik SVG Toolkit. It takes a parameter that acts as a switch to turn the sandbox on or off. The download page on the Batik website shows several examples of how to use the library. Two set a security manager at start up: the squiggle browser demo and the rasterizer demo. While the squiggle browser demo sets a manager and never changes it, the rasterizer demo calls `enforceSecurity` with a true argument the first time and a false argument the second time, which enables then disables the sandbox. While this was an interesting occurrence, there seems to be no valid reason to disable the sandbox in this case other than to show off the capability to do so.

Ant, Freemind, and Netbeans explicitly set then change the manager during runtime. Ant allows the users to create build scripts that execute Java classes during a build under a user specified permissions set. Figure 5 shows an example permission set from the Ant Permissions website.[7] The contents of the `grant` element provide the application all permissions, but the contents of the `revoke` element restrict the application from using all property permissions. Due the use of a defenseless security manager, malicious code can easily disable the sandbox and perform all actions including those requiring `PropertyPermissions`.

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

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

Figure 6: Initialization of the field in Freemind's custom security manager that stores the proxy security manager.

When Ant is about to execute an external, constrained class, it saves the current security manager and replaces it with a custom manager. The custom manager is not initially defenseless given a self-protecting permission set, but it contains a private switch to make the manager defenseless for the purposes of restoring the original manager. With this implementation, Ant catches applications that perform actions restricted by the user while typically protecting sandbox settings. However, it is not clear this implementation is free of vulnerabilities. Netbeans similarly sets a security manager around a separate application.

Both of these cases require a defenseless security manager, otherwise the application would not be able to change the current security manager. A better implementation would use a custom class loader to load the untrusted classes into a constrained protection domain. This approach would align with the intended usage of the sandbox. Additionally, it would be more clearly correct and trustworthy while allowing Ant and Netbeans to run inside of a self-protecting sandbox.

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

Freemind attempted to implement an architecture that would allow the sandbox to enforce a stricter policy on the Groovy scripts than on the rest of Freemind. Their design

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

Figure 7: Example exploit that breaks out of the scripting sandbox in Freemind to execute arbitrary code.

centers around the use of a custom security manager that is set as the system manager in the usual manner. This custom manager contains a field that holds a proxy manager to be used during the execution of scripts. In this design, all checks to the security manager are ultimately deferred to the proxy manager set in this field. When this field is set to `null`, the sandbox is effectively disabled even though the system's manager is still set to the custom manager.

Figure 6 shows how Freemind sets the proxy security manager field. Once a manager is set, if `setFinalSecurityManager` is called again with a different security manager, a `SecurityException` is thrown, but calling the method with a reference to the set manager disables the sandbox. The comment implies this specific sequence of operations was implemented to prevent malicious applications from changing the settings of the sandbox.

The Freemind code responsible for initiating the execution of the Groovy scripts sets a proxy security manager that does not allow unsigned scripts to create network sockets, access the file-system, or execute programs on the machine. The manager explicitly allows all other permissions by overriding permission check methods with implementations that do nothing. As a result, a malicious script can turn off the sandbox at any point.

We demonstrated that the custom security manager is easily removed using reflection to show that the problem is more complex than simply fixing permission checks related to setting the security manager. Figure 7 shows a Groovy exploit to turn off the manager. The script gets a reference to the system's manager and its class. The class has the same type as the custom security manager, thus the exploit gets a reference to the proxy manager field. The field is made public to allow the exploit to reflectively `null` it, disabling the sandbox to allow "forbidden" operations.

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

5. RULES FOR FORTIFYING THE SANDBOX

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

5.1 Privilege Escalation Rule

The *Privilege Escalation rule* ensures that, if a self-protecting security manager is set for the application, a class may not directly load a more privileged class. This rule is violated when the protection domain of a loaded class implies a permission that is not implied in the protection domain that loaded it. About half of recent exploits break this rule to elevate the privileges of their payload class.

If all classes in the Java Virtual Machine (JVM) instance were loaded at the start of an application, this rule would never be broken. However, the JVM loads certain classes on demand, and some of the JVM classes have the full privileges. Classes in packages that are listed in the `package.access` property of `java.security.Security` are not subject to this rule because they are intended to be loaded when accessed by a trusted proxy class.

5.2 Security Manager Rule

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

6. VALIDATING THE RULES

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

In this section, we evaluate the protection merits and backwards-compatibility of the rules presented in Section 5 through an implementation of runtime monitors that enforce them. This evaluation was done in collaboration with a large aerospace company.

Section 6.1 discusses how we implemented our runtime monitors using JVMTI. Section 6.2 explains the methodology behind and results of an experiment we conducted to determine how effective the rules are at stopping existing exploits without breaking benign applications. Finally, Section 8.2 covers prior work related to Java exploit mitigations.

6.1 Implementation Using JVMTI

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

Our agent [8] must intercept three events to enforce the Privilege Escalation and Security Manager rules: `ClassPrepare`, `FieldAccess`, and `FieldModification`. Enforcement of these rules is discussed in subsections 6.1.1 and 6.1.2.

The field events require JVMTI to turn off the JIT, which slows down program execution enough that our monitors are not suitable for adoption on their own. JVMTI implementations can avoid this limitation, but avoidance would likely

[8]Our agent is open source. An anonymized version of the tool can be found at http://goo.gl/In6Di0

increase implementation complexity beyond what is reasonable for a diagnostic interface. We are currently in communication with the OpenJDK developers on their security-dev mailing list regarding enforcement of our rules in the JVM itself to avoid overhead issues. ***Todo 6-1:*** *Note: I moved the next sentence from the intro, which was too long. It may not belong here and definitely needs to be integrated into the surrounding text.* This implementation strategy is motivated by two concerns: Existing mechanisms for monitoring the execution of Java applications are either (1) insufficient to securely enforce the rules (e.g. bytecode instrumentation) or, like JVMTI, (2) unacceptably degrade performance by disabling the just-in-time compiler (JIT). [9]

6.1.1 Enforcing the Privilege Escalation Rule

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

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

To enforce the Privilege Escalation rule, our agent registers for the `ClassPrepare` event, which allows the agent to inspect a class after it is fully loaded but just before any of its code is executed. Assuming the loaded class is not in a restricted-access package, the agent inspects the stack frame to determine which class caused the new class to be loaded. The agent must then get the protection domains for both classes to ensure the loaded class is not more privileged.

6.1.2 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 is accessed via `System.getSecurityManager()` and modified using `System.setSecurityManager()`, but we must monitor the field instead of instrumenting these methods to detect type confusion attacks.

The agent stores a shadow copy of the application's most recent security manager to have a trusted copy of the manager that can be used to check for rule violations. This copy is only updated by the modification event, which receives the new manager as a parameter from JVMTI whenever the event is triggered.

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

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

6.2 Effectiveness at Fortifying the Sandbox

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

We ran the ten exploits in our set under the following conditions: (1) without the agent, (2) with the agent but only enforcing the Privilege Escalation rule, and (3) while enforcing both rules. We tested the Privilege Escalation rule separately from the Security Manager rule because the latter stops all of the exploits on its own. All ten of the exploits succeed against our JRE without the agent. Four were stopped by the Privilege Escalation rule. All ten were stopped when both rules were enforced. The exploits that

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

Table 4: Effectiveness test results.

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

were not stopped by the Privilege Escalation rule were either type confusion exploits or exploits that did not need to elevate the privileges of the payload class. The payload class does not need elevated privileges when it can directly access a privileged class to exploit. Table 4 summarizes our results using the specific CVEs each exploit targeted.

6.3 Limitations

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

7. THREATS TO VALIDITY

7.1 Study limitations

7.1.1 Internal Validity

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

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

7.1.2 External Validity

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

7.1.3 Reliability

While the majority of the study is easily replicable, GitHub search results are constantly changing. Using GitHub to generate a new dataset using our method would likely generate a different dataset. Furthermore, over the course of our security manager study, two applications either became private repositories or were removed from GitHub (FileManagerFtpHttpServer and Visor).

8. RELATED WORK

8.1 Study

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

8.2 Mitigation

Our rules increase the security of the sandbox by effectively removing unnecessary features. Prior work has taken a different approach, instead focusing on re-implementing the Java sandbox or adding to the sandbox to increase security. Cappos et al. created a new sandbox structure. They implemented a security isolated kernel to separate sandboxed applications from the main system [29]. They validated this structure by translating past Java CVEs into exploits for the new kernel. Provos et al. describe a method of separating privileges to reduce privilege escalation [30]. Their approach is partially implemented in the Java security model. Li and Srisa-an extended the Java sandbox by providing extra protection for JNI calls [31]. Their implementation, Quarantine, separates JNI accessible objects to a heap which contains extra protection mechanisms. The performance of their mechanism is also measured using DaCapo. Siefers et al. created a tool, Robusta, which separates JNI code into another sandbox [32]. Sun and Tan extend the Robusta technique to be JVM independent [33].

Java applets are the most common ways to transmit Java exploits. Detectors have been created to identify drive-by downloads in JavaScript [34], and in Adobe Flash [35]. Helmer et al. used machine learning to identify malicious applets [36]. Their approach monitored system call traces to identify malicious behavior after execution. However, this

approach is entirely reactive. Our approach terminates exploits when they attempt to break out of the sandbox, before the exploit performs its payload. Schlumberger et al. used machine learning and static analysis to identify common exploit features in malicious applets [37]. Blasing et al. used static analysis and dynamic analysis of sandboxed executions to detect malicious Android applications [38]. Unlike these automated approaches, our rules shows that unique mitigation strategies can be created with a better understanding of how applications interact with the sandbox.

9. CONCLUSION

Todo 9-1: *CLG moved the following two paragraphs from what used to be section 3; my thought is we could have a longer "discussion" of Java software development and its relationship to security, and I think these two paragraphs might fit in well there.* Many of the recent type confusion and privilege escalation vulnerabilities would not have been introduced if the JRE were developed strictly following "The CERT Oracle Secure Coding Standard for Java" [39]. For example, Svoboda [7, 40] pointed out that CVE-2012-0507 and CVE-2012-4681 were caused by violating a total of six different secure coding rules and four guidelines.

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

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

These findings, in combination with our analysis of recent Java exploits, enabled us to define two rules which together successfully defeated Metasploit's applet exploits without breaking backward compatibility with benign applications when enforced by an experimental JVMTI agent. Some of the studied applications used the security manager to prevent third party components from calling `System.exit()`. More generally, frameworks often need to enforce constraints on plugins (e.g. to ensure non-interference). This suggests that Java should provide a simpler, alternative mechanism for constraining access to global resources. This is supported by our findings that show developers attempting to make non-trivial use of the sandbox often do so incorrectly. One intriguing possibility is to allow programmers to strengthen the policy temporarily (e.g. by adding a permission).

We indirectly observed many developers struggling to understand and use the security manager for any purpose. This is perhaps why there were only 46 applications in our sample. Some developers seemed to misunderstand the interaction between policy files and the security manager that enforces the policy. Other developers appear confused about how permissions work. In particular, they do not realize that restricting just one permission but allowing all others enables a *defenseless* sandbox. Our concerns are shared by the IntelliJ developers, who included static analysis checks to warn developers that a security expert should check their interactions with the security manager.[10] In general, sandbox-defeating permissions should be packaged and segregated to prevent accidental creation of defenseless sandboxes. More generally, some developers appear to believe the sandbox functions as a blacklist when, in reality, it is a whitelist. These observations suggest that more resources—tool support, improved documentation, or better error messages—should be dedicated to helping developers correctly use the sandbox.

10. REFERENCES

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

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

[3] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2.," in *USENIX Symposium on Internet Technologies and Systems*, pp. 103–112, 1997.

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

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

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

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

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

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

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

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

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

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

technotes/guides/security/PolicyFiles.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[36] 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.

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

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

[39] 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.

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