Fortifying the Java Sandbox

***Abstract*—The Java sandbox attempts to allow arbitrary code to execute in way that that limits the permissions of the executing code, thus protecting the system where the code is executing. Unfortunately, exploits are continually being created that bypass the security features of the sandbox. The objective of this paper is to gain an understanding of how applications interact with the sandbox and make an initial attempt to fortify the sandbox against future exploits. To achieve those results, we started with an investigation of how 46 open source applications interact with the sandbox. The outcome of this investigation supported the hypothesis that we can could restrict behavior of an application to increase security. Two initial fortification strategies were then created and verified with 10 Java exploits from the past 2 years. The effects of the fortification strategies were also performance tested on the Dacapo Benchmarks to determine the strategies' effects on running applications. The study reveals that a significant number of exploits can be stopped by providing extra strength to the sandbox.**

I. Introduction

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. Unfortunately, recent exploits have shown that this security goal has not been achieved. Instead of the sandbox completely protecting the application, security researchers are constantly finding ways to exploit how the sandbox is implemented. Security Exporations, a company that research security vulnerabilities in applications, found that the Java Virutal Machine (JVM) contained 19 weaknesses and developed 12 expoits to demonstrate these weaknesses in a report on April 2nd 2012 (&& add a citiation of http://www.security-explorations.com/en/SE-2012-01-press.html if you keep this sentence &&). Hackers have also discovered zero-day exploits, with the exploit in August 2012 and January 2013 making widespsread news and detailed in a CERT blog post(cite: http://www.cert.org/blogs/certcc/post.cfm?EntryID=136 - also not 100% happy with this sentence at the moment).If this trend continues, regular updates will not be enough to ensure the integrity of Java applications.

Past investigations of Java security exploits have shown Java malware commonly alters the sandbox's settings. This alteration of the sandbox is usually by turning the sandbox off with a call to `System.setSecurityManager(null)`, since the `SecurityManager` is how the application interacts with the sandbox.

This knowledge leads to the question: can malicious applications be uniquely deteced and stopped at runtime based on how they interact with the sandbox? To answer this question, it is imporant 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, verifing the results of the static analysis

Based on the findings of this analysis, two rules for fortifying the sandbox were implemented: the Privilige 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 inidividual classes and then enforcing the permissions through permissions checks. Figure 1 summarizes the components of the sandbox that are relevant to this work. Essentially, when a class loader loads a class from some location (e.g., network, filesystem, etc.) the class is assigned a code source. The assigned code source is used to indicate the origin of the code and to associate the class with a protection domain. Protection domains segment the classes of an application into different groups, where each group is assigned a unique permission set. The permission sets contain permissions explicilty 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 [1]). The application defines how to assign clases 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 permissable 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 malicous 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

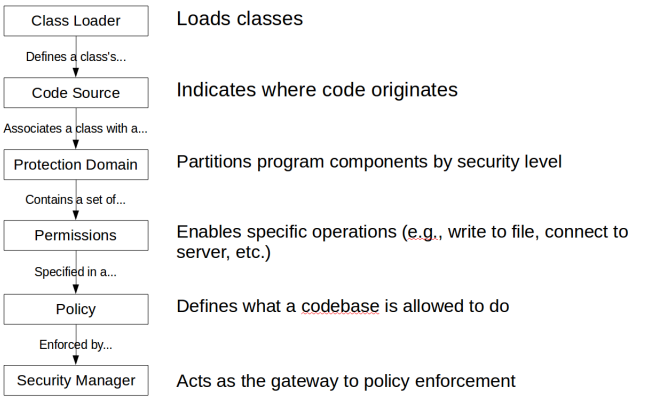

Figure 1. A high-level summary of the Java sandbox as it pertains to this work.

```
1  import java.lang.reflect.Method;
2  import java.security.AccessController;
3  import java.security.PrivilegedExceptionAction;
4
5  public class Payload implements PrivilegedExceptionAction {
6      public Payload() {
7          try {
8              AccessController.doPrivileged(this);
9          } catch(Exception exception) { }
10     }
11
12     public Object run() throws Exception {
13         // Disable sandbox
14         Class cl = System.class;
15         Method m = cl.getMethod("setSecurityManager",
16             new Class[] { SecurityManager.class });
17         m.invoke(null, new Object[1]);
18         return null;
19     }
20
21     public static void outSandbox() throws Exception {
22         // Do malicious operations
23     }
24 }
```

Figure 2. A typical Java exploit payload from http://pastebin.com/QWU1rqjf.

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 [2], [3], [4], [5], [6], [7], [8].

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 IV. 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 [9].

B. Java Exploits

Malicious drive-by downloads using Java applets as the vector were widely reported between 2011 and 2013 (&& probably should add a citiation here &&). While Java applets should prevent malicious applications from executing their payload, vulnerabilities in the Java Runtime Enviroment (JRE) were leveraged by exploits to set the secruity 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 applicaition can now execute the payload on the host system. Figure 2 shows a typical payload.

A prominant subclass of Java exploits take advantage of a confused deputy vulnerability [10], 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 [11]), 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" [12]. For example, Svoboda [13], [14] 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

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

Java sandbox without breaking backwards compatibility and not with the specifics of particular exploits.

III. METHODOLOGY

In security, the best mitigation strategy is not only patching individual vulnerabilies once the vulnerabilites have been discovered, which in many cases could mean reacting after significant damage has already occured. Instead, a better approach to security is to be proactive, for example classifying and catching a new exploit before it can do serious damage. To successfully classify a new exploit, the key features which seperate an exploit and a benign application must be determined. Attempting to catch exploits by the specific vulnerability used would be difficult, because this approach likely requires being aware of the vulnerability before the exploit. However, studies into Java exploits suggest that Java exploits interact with the sandbox in a similar manner. This knowledge leads one to question if exploits can be seperated from benign applications using the way the applicaion interacts with the sandbox.

In attempt to study how bengin applications interacted with the sandbox, we investigated how applications interact with the security manger, since the security manager is the part of the sandbox which handles how applications interact with the sandbox. To investigate how benign applications interact with the security 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:

Weakest Claim: *Benign applications do not disable the sandbox.* If this claim is shown to be true, exploits can be differentiated from benign applications by number of attempts to turn off the sandbox. A mitigation strategy using this claim would easily be able to classify an unknown application. However, exploits which weaken the sandbox but do not disable the sandbox, would not be caught. For example, attackers could bypass the mitigation strategy by either weakening the policies enforced by the manager or replacing the current manager with one that never throws a `SecurityException`.

Weak Claim: *Benign applications do not weaken the sandbox.* The goal of this claim is to be able to classify benign applications and exploits by the changes to the sandbox. A mitigation using this claim would prevent weakening the sandbox during runtime and thus catch both exploits which turn off the sandbox and exploits which only weaken the sandbox. However, an implementation of this mitigation would require differentiating between changes which weaken the sandbox and changes which do not weaken the sandbox. Classifying changes to the sandbox 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.* This claim would allow seperating exploits from benign applications if the sandbox is changed while the application is using the sandbox for security. A mitigation strategy using this claim would allow clear differentiation between benign applications and exploits, while also preventing exploits from changing the sandbox in any way. Implementing this claim requires determining if the secuity manager is self-protecting, which can be easily acheived at any time.

Strongest Claim: *Benign applications do not change the sandbox.* The goal of this claim is to be able to classify benign applications and exploits using the seperator that only exploits change the sandbox during execution. If the study supports this claim, any attempted change to the security manager could be used to determine an exploit attempt. A mitigation strategy using this claim could clearly differentiate betwen benign applications and exploits, while also preventing the sandbox from changing in any way.

Our emprical analysis used applications from the Qualitas Corpus (QC) [15] 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 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. The 29 applications were then inspected to make sure they were the latest released version of the application. If the application contained a newer release than the version already in Qualitas, the application was updated to the latest released version. We performed a similar process using the GitHub search feature configured search through Java files for the SecurityManager keyword. Initially, we extracted the top 6 applications which used the term SecurityManager, but upon initial inspection, we came to find this filtering method was producing a high false postive rate, meaning that developers were using the word SecurityManager in ways that were unrelated to java.lang.SecurityManager (later in-

depth inspection determined 4 out of the 6 applications did not use the java.lang.SecurityManager). We then refined our search to only include applications which seemed to set up a sandbox. Thus we extracted the top 7 applications which used the term System.setSecurityManager(). To make sure we covered applications which disabled the security manager, the top 7 applications where extracted which used the term System.setSecurityManager(null). Upon inspecting the extracted results, 2 applications were already covered in the Qualitas Corpus so they were removed from the dataset. We also removed an application which was written in Ruby but contained Java files for a final total of 17 applications in the Github dataset. All applications that were downloaded from Github were downloaded as the latest commits so the Github applications did not need to be updated.

We created static and dynamic analysis tools to assist the manual inspecton of applications. Our static analysis tool consisted of a plugin for FindBugs [16], a tool used to statically analyze Java programs for code that matches a given bug pattern. Our plugin used the dataflow analysis in Findbugs to determine the lines in the application where `System.setSecurityManager()` was called, as well as the lines where the argument used in the setSecurityManager call was initialized. We also created a dynamic analysis tool using the Java Virtual Machine Tool Interface (JVMTI)[1]. JVMTI is desgined to allow tools to inspect the current state of Java applications and allow tools to control the execution of the Java application. JVTMI is commonly used for Java debugging and profiling tools. Our dynamic analysis tool set a modification watch on the `security` field of Java's `System` class, similar to the way debuggers watch a variable. The security field of the System class was choosen because the security feild is the variable which holds the current security manager object for the application, and thus the current sandbox object. The field watch printed out the class name, source file name, and line of code where any change that effected the security field took place. The printed message contained a special notice if the security field was set to `null`.

We split the applications that used the security manager between two reviewers. The reviewers both analysed applications in using the same steps, which are listed below.

1) The reviewer ran grep on all the Java files in the application to output the lines which contain the word SecurityManager.
2) If it was possible to quickly determine the application only used the SecurityManager in comments or in ways that were unrelated to java.lang.SecurityManager, the reviwer labeled the application as a false postive.
3) If the application was not a false positive and could be compiled, the reviewer ran the Findbugs plugin on the application to highlight where the application sets the security manager for the system (turns on the sandbox).
4) The reviewer manually inspected the lines mentioned in the static analysis. Starting with the line where the system security manager was set and tracing the code to where the security manager was initialized.
5) The reviewer then 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 applications that were not a false positive, could be compiled, and effected the security manager during the execution of the application, the reviewer monitored the applications with the dynamic analysis tool while executing the applications in ways that effect the security manager. This step verified the conclusions made from previous steps
7) Finally, the reviewer summarized the interaction between the application and the security manager with an emphasis on points that support or invalidate each claim.

As a way to ensure the analysis steps produced the expected results and that both reviewers produced similar reviews for the same application, the analysis started with both reviewers independently analyzed the same 6 applications. These independ provided feedback to the reviewers, allowing the reviewers to focus on the same interactions in the application as well as notify the reviewer if he was missing something imporant.

IV. Research Questions and Results

A. How do applications interact with the SecurityManager?

The first goal of our inspection of Java applications was to gain an understanding of how Java applications interacted with the SecurityManager. To do so, applications were divided into categories based on how the application interacted with the SecurityManager. The categories were: setting a SecurityManager and then changing the application's SecurityManager during execution, setting a SecurityManager and not changing it during execution, interacting with a SecurityManager but never setting one, and not interacting with a SecurityManager at all. The first category, setting a SecurityManager and then chaning the SecurityManager during execution, means that the applications either sets a different SecurityManager as the SecurityManager for the system, or the application alters the current SecurityManager for the system. Since this category was the most likely category to affect the validity of the claims, the applications in this category are further explained in later sections of this paper. The second category, setting a SecurityManager and not changing it during execution, meant that for each execution path, there was at most one place the application set a SecurityManager. This category supported our claims unless the SecurityManager was explicitly set to null, which would mean an attempt to disable a SecurityManager if one was set previously. Only one application in this category (JTimelag), explicilty set the SecurityManager to null. This situation is discussed futher in the section IV-B2. The third category, interacting with a SecurityManager but never setting one, consists of applications which contained code indicating the application was designed to work inside of a sandbox but never explicitly set up a sandbox in the application. The indicating code consists of adding extra permission checks if the application was inside of a sandbox, (**may want to explain doPrivileged here**) using doPrivileged calls to avoid checking the permissions of all classes on the stack, or ensuring that sections of the application would work within

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

a sandbox using test cases. This category was further subdivided into applications which altered execution inside the main application (meaning extra permission check or calling doPrivileged) and applications which only contined test cases that interacted with the sandbox. The only contained test cases category also includes one application, system-rules, which was a utility for test cases. The final category, not interacting with a SecurityManager at all, means that the application either explicitly referenced the SecurityManager in comments but not in the source code or the application defined a class containing the name SecurityManger but the defined class did not extend java.lang.SecurityManager in any way.

After inspecting the 29 applications in the filtered Qualitas dataset, the applications were classified based on how they interacted with the SecurityManager. (## probably should make this a table ##) Five of the applications set and then changed the SecurityManager as the program was running. Six of the applications contained code that set a SecurityManager, but did not change the SecurityManager after it was set. Thirteen applications contianed code that indicated they were designed to run inside a SecurityManager. Of those thirteen applications, ten contained code in the application that altered behavior when a SecurityManager was set. Three of those thirteen applications only interacted with a SecurityManager in test cases. The remaining five applications in the filtered Qualitas dataset were false positives.

The GitHub dataset also contained similar results. Three of the seventeen applications set and then changed the SecurityManager. One application set a SecurityManager and then did not change the set SecurityManager as the program was running. Eight applications contained code that indicated they were designed to work inside a sandbox if one was set. Of those eight applications, three altered execution if a sandbox was set while five of the eight applications only interacted with a sandbox in test cases. The other five applications were false positives.

B. Non-security uses of the Sandbox

When investigating the applications in both of the datasets, the first surprising discovery was that applications used in 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.

1) 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 application 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
692 ...
703 getCurrentProject().executeTargets(targets); \\Note: Ant is execute
704 ...
703 finally {
704 ...
725
726 if (System.getSecurityManager() instanceof AntSecurityManager) {
727 System.setSecurityManager(originalSM);
728 }
```

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.

2) Reducing Complexity In Web Application Development: 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 permissio
57
```

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.

C. 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 appications 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:

```
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      ...
```

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 demo can be set to call enforceSecurity with a true arguement 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 occurance, there seemed to be no valid reason, other than showing off the capabilties 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 dynmically 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 also tried to solve a similar problem, but demonstrates the difficulty of currectly 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 (*** should put version somewhere ***), 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 setting and chaning 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 WildfiySecurityManager 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 assiging the privileged classes the AllPermissions and then executing doPrivileged on the privileged actions that the privileged class needs to do.

D. Rules

Based on the result of our investigation, it was determined that we could restrict applications in two ways to improve security, preventing applications from loading privileged classes when the loading class did not have the required itself and preventing changing the SecurityManager when a *self-protecting* SecurityManager is set. These restrictions were classified as rules, the privilege escalation rule and the SecurityManager rule.

1) Privilege Escalation Rule: The privilege escalation rule states that a class may not load a more permissive class if a SecurityManager is set for the application. This means that the loading classes permissions has to be less restrictive or the same level of restriction as the loaded class. 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 be without exceptions. 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 excpetion for these classes. Specifically, the rule makes exceptions for classes returned by the call `java.security.Security.getProperty("package.access`

2) SecurityManager Rule: The SecurityManager Rule states that a SecurityManager cannot change if a *self-protecting* SecurityManager has been set by the application. By setting a *self-protecting* SecurityManager, 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.&&)

V. 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 IV 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 [17], [18], [19], [20], and the machine learning community has put some effort into detecting exploits delivered via drive-by-downloads using Java applets and similar technologies [21], [22], [23], [24]. We implemented a tool in JVMTI to proactively stop exploits written directly in the Java programming

language to exploit vulnerable Java code[2]. In particular, our tool focuses on fortifying the Java sandbox to reduce the probability that the sandbox will 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 analysers. 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 [19] 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 [17], our tool could be isolated to its own security layer with access only to the information it gets from JVMTI. We did not attempt to apply these solutions because the required tools and code are not publicly available, which would make it difficult, if not impossible, for most people to adopt our tool.

Our agent may be configured to run in enforce or monitor mode. In enforce mode a violation of either rule causes the agent to log the offending behavior and terminate the JVM to which the agent is attached. 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 3 shows an example of a popup displayed after an exploit was caught breaking the privilege escalation rule.

[2]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.

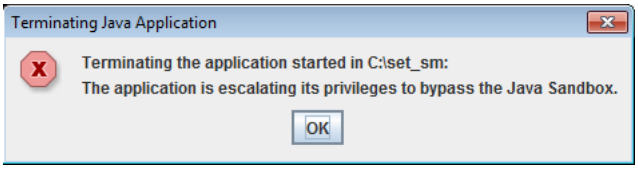

Figure 3. A popup from our agent after it caught an exploit breaking the privilege escalation rule.

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[3] 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[4] 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`

[3]`Class.getProtectionDomain()`

[4]`RuntimePermission("getProtectionDomain")`

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 IV. 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. Performance

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[5] on 64-bit Windows 7 against the initial release of version 7 of the JRE. This version of Metasploit contains twelve applets that are intended to exploit JRE 7 or earlier, but two did not successfully run due to Java exceptions we did not debug. Metasploit contains many Java exploits outside of the subset we used, but the excluded exploits either only work against long obsolete versions of the JRE or are not well positioned to be used in drive-by-downloads.

We ran the ten exploits in our set under the following conditions: without the agent, with the agent but only enforcing the privilege escalation rule, and while enforcing both rules. We ran these conditions to respectively: establish that the exploits succeed against our JRE, test how effective the privilege escalation rule is without the security manager rule, and evaluate how effective the agent is in the strictest configuration currently available. Running the privilege escalation rule alone is interesting because it provides some indication of how effective this rule is at stopping applet exploits in general, outside of simply fortifying the security manager. 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 II summarizes our results using the specific CVE's each exploit targeted.

D. Limitations

Unfortunately, neither of these rules will able to stop 100% off 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

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

Table II
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

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 beleive the rules created in this study provide a substanial 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 techiques.

VI. 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) Correctly implementing a SecurityManager in an application is easily done incorrectly, 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 likely not inclusive of all Java 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] "Permissions in the JDK," 2014.
[2] 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.
[3] 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.
[4] E. W. F. D. S. Wallach, "Understanding Java Stack Inspection," pp. 52–63, 1998.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] "Security Vulnerabilities in Java SE," Technical Report SE-2012-01 Project, Security Explorations, 2012.
[10] N. Hardy, "The Confused Deputy: (or Why Capabilities Might Have Been Invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988.
[11] "Vulnerability Summary for CVE-2012-4681," Oct. 2013.
[12] 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.
[13] D. Svoboda, "Anatomy of Java Exploits."
[14] D. Svoboda and Y. Toda, "Anatomy of Another Java Zero-Day Exploit," Sept. 2014.
[15] 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.
[16] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.
[17] 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.
[18] 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.
[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.