Fortifying the Java Sandbox

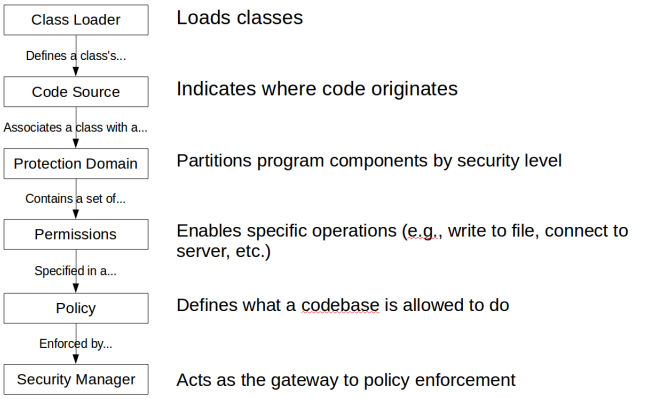

Figure 1. A high-level summary of the Java sandbox as it pertains to this work.

I. Introduction

II. Background

The Java sandbox was designed to enable the safe execution of code from untrusted sources. 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 code source is used to indicate the origin of the code while it is loaded in the JRE and to associate the class with a protection domain. Protection domains segment classes into different groups that are assigned a security policy. Each domain imposes one policy on the classes it contains where a policy includes a set of permissions that grant the contained classes the ability to perform operations such as writing to the filesystem, accessing the network, using certain reflection features, etc. (see a more complete list at [1]). Each JRE contains a default policy configuration that grants applications from the local filesystem all permissions and applets from the web a strict set of permissions that prevent them from performing malicious operations on the host system.

To enforce the security policy, a security manager must be set (typically by calling `System.setSecurityManager(new SecurityManger())`). The security manager acts as the gateway to policy enforcement in the Java sandbox. Whenever Java code wants to ensure a caller has the correct permissions to execute protected code, the callee must query the security manager. For example, if an application attempts to use a JRE class to write to a file (e.g Java.io.FileOutputStream), the JRE class will ensure that the caller has permission to write to the file before performing the write. When such a query is initiated, the security manager walks the stack frames for the current calling context to ensure every class in the context has the correct permissions or a privileged caller has marked the context as privileged. If the context is not marked as privileged and even one class in the stack frame does not have the correct permissions, the manager will throw a SecurityException. Stack-based access control is discussed in more detail in [2], [3], [4], [5], [6], [7], [8]. Typically a security manager is automatically set by the JRE when a class is loaded using class loaders for sources that are nearly always untrusted such as applets.

Malicious drive-by downloads using Java applets as the vector were widely reported between 2011 and 2013. While the security manager should prevent applets from performing operations that are useful to a malicious actor, vulnerabilities in the JRE were leveraged to set the manager to null. The latter operation effectively disables the Java sandbox, thus allowing previously constrained classes to perform any operation that the JRE can perform. These vulnerabilities tend to fall into one of two categories: those that exploit a type confusion vulnerability to set the security manager to null and those that exploit a confused deputy vulnerability [9] in a privileged JRE class to null the security manager.

In the case of type confusion a vulnerability is exploited to break type safety, thus allowing the attacker to craft an object that can perform operations as if it is an instance of a class of a different type or as if it is an already existing object of a different type. For example, attackers will craft objects that either (1) point to the System class to cause any operation they perform to happen on the real System class, thus allowing them to directly alter the field where the security manager is stored or (2) act as if they have the same type as a privileged class loader to load a payload class with all permissions (see CVE-2012-0507 [10]). In the case of a confused deputy, the exploit often attacks a privileged class that is accessible by any class to obtain an instance of a vulnerable privileged class that is only accessible to other privileged classes. The restricted-access class is targeted because it contains a vulnerability that can be leveraged to execute privileged code, typically to null the security manager (see CVE-2012-4681 [11]). While it is possible for an accessible privileged class to contain a vulnerability that will allow an attacker to null the manager more directly, these classes are typically implemented more carefully to avoid these types of issues.

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

For the purpose of fortifying the sandbox, we distinguish between self-protecting and defenseless security managers. A *self-protecting* security manager enforces a policy that prevents encapsulated classes from changing the manager or the policy it enforces. In contrast, a *defenseless* security manager allows encapsulated classes to change the manager or the enforced policy. A defenseless manager does little to improve the security posture of the encapsulated Java application. Table I summarizes the set of permissions used to distinguish between self-protecting and defenseless security managers. A manager enforcing a policy that contains even one of the listed permissions is considered to be defenseless. A subset of the permissions in this list were identified in [15].

III. Methodology

While Java exploits tend to follow one of several high-level patterns, the individual vulnerabilities they exploit are diverse in their form and their location in the JRE. Given this complexity, we must understand how benign Java applications interact with the Java sandbox to design justifiable mitigations. The security manager is the means by which applications interact with the Java sandbox, thus we undertook an empirical analysis consisting of static, dynamic, and manual inspections of the open source Java application landscape to investigate benign uses of the security manager. Our empirical analysis aimed to validate the following hypotheses, roughly categorized by the strength of the mitigation that is possible if the hypothesis is true:

Weak Hypothesis: *Benign applications do not disable the security manager.*

Moderate Hypothesis: *Benign applications do not weaken the security manager by allowing previously disallowed operations.*

Strong Hypothesis: *Benign applications do not change the security manager or the enforced policy in any way if a self-protecting security manager has been set.*

If the weak hypothesis is true, we can monitor applications to ensure they do not set the manager to null. This would prevent exploits as they are currently written, but it would be simple for attackers to shift to weakening the policies enforced by the manager or to replace the current manager with one that never throws a SecurityException. Successful validation of the moderate hypothesis would be stronger because it would prevent the manager and policy from being weakened, but the implementation of this mitigation would be potentially problematic because one would have to determine when a permission can be changed without creating a security vulnerability. This is difficult because it requires context a tool may not have. For example, if a permission to write to a file is replaced by a permission to write to another file, how does a tool know if the policy is weaker or not? The strongest mitigation is possible if the strong hypothesis is true because it achieves a similar effect to the moderate mitigation while being substantially easier to implement.

We turned to the Qualitas Corpus (QC) [16] and GitHub to form a dataset of applications that use the security manager. To sort out relevant 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 reduced the overall set of applications to inspect from 112 to 29. While this led to the inclusion of applications that had the keyword in comments or in unit tests that would not appear in production code, these applications were easily sorted out in later steps. We performed a similar process using the GitHub search feature configured to show source code, Java applications, and the most relevant results. At the time of our search, GitHub did not group results by project, thus we searched through the first 20 pages and found 17 unique projects that used the security manager to add to the dataset. While the page limit meant we only looked at a few hundred of the nearly 100,000 files returned, we believe this procedure was sufficient to ensure our dataset is representative of how open source Java applications use the security manager.

We created static and dynamic analysis tools to further sort through the dataset before manually inspecting projects. Our static analysis is a plugin for FindBugs [17] that uses a dataflow analysis to determine what initialized security managers may be set as System security manager by calling `System.setSecurityManager`. We created a dynamic analysis using JVMTI[1]. Our dynamic analysis set a modification watch on the `security` field of Java's System class. The security field is a private and static field that stores the SecurityManager object that will be used by all JRE classes when making security policy decisions for the running Java application. The watch printed out the class name, source file name, and line of code where any write to the security field took place. The printed message contained a special notice if the security manager was being set to `null`. After running both of these tools on each project we were armed with the information required to further narrow our manual inspection efforts: which applications change the security manager, where they make the change, and where SecurityManager objects are initialized.

We split the applications that used the security manager between two reviewers. The reviewers utilized a checklist to ensure that they consistently inspected each application. Table II provides a summary of the items on our checklist; the full list provides details on carrying out each step. After performing a manual inspection, the reviewer ran the dynamic analysis on the application again, but this time with an emphasis on running executions that were known to exercise the security manager to ensure we didn't miss an important execution. Finally, the reviewer wrote a summary of how the application used the security manager with an emphasis on points that support or reject each hypotheses.

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

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

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

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.

Table II
SUMMARY OF CHECKLIST FOR MANUAL REVIEWS IN THE EMPIRICAL ANALYSIS OF SECURITY MANAGER USAGE.

Identify call sites for setSecurityManager
Identify references to SecurityManager objects that reach a call site
Identify possible values for SecurityManager objects that reach a call site

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 hypotheses, 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 hypotheses 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 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.currentThread()));
692 ...
703 getCurrentProject().executeTargets(targets); \\Note: Ant is executed here
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.

(**** may need to move the next paragraph in the future ****)

One example of the difficulty of creating a sandbox that can be turned on and off dynamically was found in Freemind. 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 ***).

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

-talk about how applications provide the option to set and remove security managers, also give an example

- talk about how Ant restricts the applications for a small section

- can talk about wildflySecurityManager

- talk about batik and the demo where they set two SMs

V. Mitigations

In section III we discussed (1) three hypotheses 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 hypothesis 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 highly targeted employees.

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 [18], [19], [20], [21], and the machine learning community has put some effort into detecting malicious applets and drive-by-downloads using similar technologies [22], [23], [24], [25]. We implemented a tool in JVMTI to pro-actively stop exploits written directly in the Java programming language to exploit other Java code[2].

JVMTI is a native interface that is intended to be used to create analysis tools for Java such as profilers, debugging, monitoring, and thread analysis tools. 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 contains a rich API, hooks for instrumenting the bytecode of loaded classes, and provides 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. 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 name 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 [20] 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 [18], our tool could be isolated to its own security layer with only access 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 use our tool.

Our agent may be configured to run in enforce mode or in 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 simply 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 assuming the agent has been configured to show popups (this was made configurable to prevent popups on headless servers). Figure 2 shows an example of a popup from when 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.

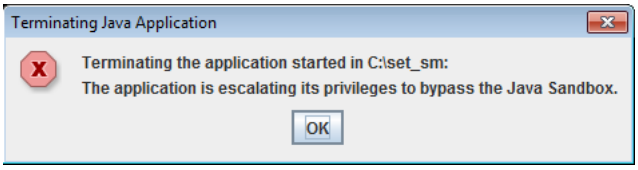

Figure 2. A popup from our agent after it caught an exploit breaking the privilege escalation rule.

Enforcing the Privilege Escalation Rule: To enforce the privilege escalation rule we must ensure that if 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 those that are public but that are not intended to be directly used by typical Java applications because they are meant for internal JRE use only. These packages are listed in the `package.access` property in the `java.security.Security` class. The only ways to directly access packages listed in this property are to exploit a vulnerability in a class that can access them or to allow access via the `accessClassInPackage` permission, but allowing this permission for the protection domain that encapsulates a Java application's classes would make the security manager defenseless (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.

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 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` 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 class 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 registering `FieldModification` and `FieldAccess` events for the `security` field of the `System` class. These events notify our agent every time the field is written to or read by any Java code. 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 violations of this rule. In a typical deployment, the agent is loaded by a JVM before the hosted Java application's code has begun executing, but if 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 the JVM.

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. 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 and the class itself even though writing to the field in one class also updates it in the other class. 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

[3] `Class.getProtectionDomain()`
[4] `RuntimePermission("getProtectionDomain")`

by the modification event.

B. Effectiveness in Stopping Exploits

C. Performance

D. Limitations

How to write exploits now...

VI. Conclusion

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.