JMD: A Hybrid Approach for Detecting Java Malware

Adrian Herrera **Ben Cheney**

Defence Science and Technology Organisation
Adelaide, Australia
Email: {adrian.herrera, ben.cheney}@dsto.defence.gov.au

Abstract

With the rapid rise in the number of exploits targeting the Java runtime environment, new tools are required to detect these malicious Java applications. This paper proposes one such tool, the *Java Malware Detector* (JMD). JMD takes a hybrid approach that combines symbolic execution, instrumentation and dynamic analysis to detect malware that subverts Java's access control mechanisms. Using this approach, we aim to derive any trigger conditions that may exist before instrumenting and executing the malware in a controlled environment to observe whether it escapes the Java security sandbox. A key element of this approach is our use of existing open-source software platforms—specifically, Java Pathfinder and AspectJ. By using real-world Java malware samples we are able to evaluate the effectiveness of JMD. The results of this evaluation show that JMD's instrumentation and dynamic analysis capabilities provide an effective tool for detecting a wide range of Java malware: we successfully detected malware variants that represent fourteen of the known access control-related CVEs disclosed over the past four years. However, our success in using symbolic execution to derive trigger conditions was limited, mainly due to the incomplete state of the `String` handling implementation in Java Pathfinder's symbolic execution plugin.

1 Introduction

The number of exploits targeting the Java Runtime Environment (JRE) has been increasing at an alarming rate. During the 12 months from September 2012 to August 2013, a Kaspersky Lab report claimed to have detected over 14.1 million attacks that relied on a Java exploit—an increase of 33.3% on the previous twelve months [9]. Cisco found that Java exploits represented 91% of all Indicators of Compromise (IoC) in 2013 [4]. Figure 1 illustrates the recent escalation of the Java malware problem in terms of the number of JRE CVEs issued each year.[1]

The most common delivery method for these exploits is a Java *applet* [9]. A Java applet is a Java application that is accessed via a web browser and executed on the local host in a Java Virtual Machine (JVM) instance. To execute a Java applet, a web browser requires the installation of a Java plug-in. Java's pervasive install base[2] means that there is a high chance a user has a Java plug-in installed, making Java a popular target for drive-by download attacks (where the user either unwittingly executes a malicious applet or authorises its execution without fully understanding the risks). Furthermore, Java exploits can provide a relatively cheap path to reliable remote code execution compared with exploits that require address space layout randomisation (ASLR) and data execution prevention (DEP) to be bypassed [34]. These combined properties have made Java a popular target for malware authors.

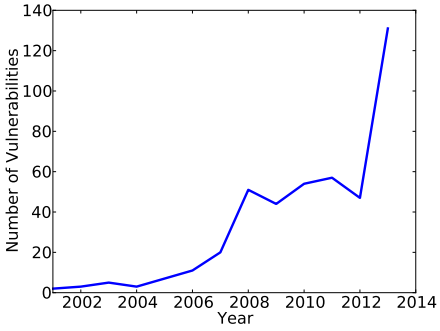

Figure 1: Sun/Oracle JRE vulnerabilities from 2001 to 2013

The prevalence of Java malware means that accurate and timely detection methods have become crucial elements of effective computer network defence. Traditional malware detection (e.g. that provided by an anti-virus product) relies heavily on signature-based techniques. However, the weaknesses of signature-based detection techniques are well-known [17, 15]—they can only detect previously discovered threats and are unable to detect zero-day exploits. Additionally, malware authors can often evade anti-virus products through obfuscation.

To supplement signature-based detection methods, dynamic analysis techniques are often used to run malware in a controlled environment (a *sandbox*) where it can be monitored for malicious activity. While dynamic analysis may overcome the limitations of signature-based detection in some cases, it is still possible for malware to evade detection by remaining dormant until a particular trigger condition is met [1, 16]. In order to identify and solve trigger conditions within a given sample (so that code coverage is maximised during dynamic analysis), symbolic execution can be utilised to explore alternate code

[1] The statistics for this graph are sourced from the CVE Details website, `http://www.cvedetails.com`.

[2] The Oracle Java 7 Windows installer claims that "three billion devices run Java".

paths and attempt to derive trigger conditions that are required for a particular code path to execute [1].

We have combined the techniques of symbolic execution, instrumentation and dynamic analysis in developing a system aimed specifically at detecting Java malware. Our Java Malware Detector (JMD) takes Java bytecode as input, performs symbolic execution on the bytecode to derive the trigger conditions required to maximise code coverage, instruments the bytecode and then performs dynamic analysis on the instrumented sample[3] to determine if the JRE's access control mechanisms have been subverted. Note that JMD is not designed to detect Java malware that targets native code vulnerabilities (§3.1 discusses our rationale for taking this approach).

The main contributions made in this paper are:

- The design and implementation of a hybrid detection system. This detection system combines symbolic execution, instrumentation and dynamic analysis techniques to specifically target Java malware.
- The extension of Java Pathfinder's symbolic execution engine (Symbolic Pathfinder) to find trigger conditions in Java malware.
- The design and implementation of a mechanism for detecting the subversion of Java's access control mechanisms using Aspect-Oriented Programming (AOP) techniques.

The remainder of this paper is organised as follows: §2 provides background information relevant to the design of JMD; §3 discusses our design and implementation decisions for JMD; §4 evaluates JMD's performance; §5 examines previous work in the area of Java malware detection; and §6 concludes the paper and suggests some ideas for future work.

2 Background

This section provides background material on the Java Runtime Environment (JRE), its security model (with a particular focus on access control mechanisms) and examples of how JRE vulnerabilities have been exploited in the past.

2.1 The Java Runtime Environment

'The JRE' (as defined by Oracle [24]) formally consists of the JVM and all associated libraries and components which enable the execution of applications written in the Java programming language. Notionally, the runtime environment for a Java program can also be abstracted into a number of layers, as illustrated in Figure 2 (adapted from [31]).

A Java application (the top layer) is compiled into machine-independent Java bytecode, which executes in a JVM (second layer) instance. As is suggested by Java's "write once, run anywhere" mantra, the JVM—which is available on several operating system (OS)/architecture combinations—allows a Java application to be cross-platform and portable. Java applications can be deployed in a variety of ways, but Java malware typically takes the form of an *untrusted* Java applet[4] (see §2.2.2 for more on the nature of untrusted applets).

[3] Dynamic analysis may proceed multiple times, depending on the number of trigger conditions identified during symbolic execution.

[4] Given that applets are the most common deployment vector for Java exploits [9], we use the terms *Java application* and *Java applet* interchangeably when discussing Java malware.

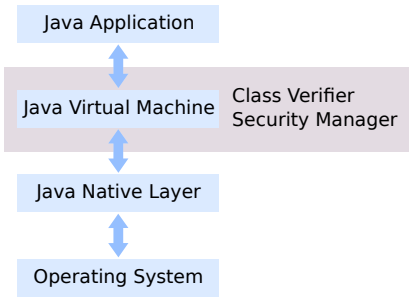

Figure 2: Runtime environment of a Java program, with some elements of the Java security architecture highlighted

Beneath the JVM lies the native layer, with which the JVM interacts when Java applications and APIs need to interface with native code. Java's APIs call native layer code in order to make platform-specific system calls (e.g. for implementing operations on a local file system) and to interact with software written in another language (e.g. C, assembly, etc.).

2.2 Code Verification and Access Control

Java's security architecture is realised by a set of language features and APIs which encompass areas such as cryptography, PKI, authentication and secure communication [23]; however, the features and APIs most relevant to a discussion of Java malware are those related to *code verification* and *access control*.

Java bytecode is verified at load-time to "ensure that only legitimate bytecodes are executed in the Java runtime" [23] (discussed further in §2.2.1). At run-time, the access control APIs mediate access to sensitive resources and operations (e.g. access to local files, sending/receiving arbitrary data over a network, etc.) in accordance with a defined security policy (discussed further in §2.2.2).

It is important to note that code execution in the native layer occurs outside the purview of Java's access control APIs. As such, Java malware seeking to subvert Java's security restrictions may choose to target either a vulnerability in the access control API (i.e. code being executed in the JVM layer) or a vulnerability in a native library called by Java (i.e. code being executed in the native layer).

2.2.1 Class Verifier

When Java classes are loaded by the JVM, the class verifier performs several passes over the bytecode in order to ensure the correctness of the class. This includes checking for forged pointers, stack overflows and underflows, access (public/protected/private) violations and ensuring type safety. If any of these checks fail, an error is thrown by the JVM.

CVE-2012-1723 [21] is an example of a vulnerability in the class verifier which allows malicious bytecode to perform a type-confusion attack. This attack is possible due to an invalid optimisation in the class verifier when a field access operation is performed [27]. Class verifier exploits such as CVE-2012-1723 are considered outside JMD's scope.

2.2.2 Security Manager

The security manager (represented by an object of type `SecurityManager`) mediates all access control decisions for Java APIs. This ensures that an

application adheres to a particular security policy at run-time.

When a local Java application is loaded from disk and run, it is executed with *full* user privileges and *without* a security manager [23, 7] (unless the user explicitly installs one in their application code or supplies the `-Djava.security.manager` command-line flag). By contrast, untrusted applets (which are typically accessed over the Internet from an unknown source) execute in the presence of a security manager, which enforces a *reduced* set of privileges in order to prevent the execution of unsafe operations. The restricted environment in which untrusted applets execute is commonly known as the *Java sandbox*.

Whenever a potentially unsafe operation is attempted by a sandboxed Java application or API, the `SecurityManager` object checks whether the class has been assigned the relevant permission (represented by a `Permission` object). The `AccessController` class is the `SecurityManager`'s mechanism for checking these permissions. If the operation is not allowed in the current Java sandbox, a `SecurityException` is thrown. Examples of potentially unsafe operations that require a particular set of permissions include file (e.g. read, write, etc.), socket (e.g. connect, accept, etc.) and `ClassLoader` (e.g. create) operations.

The set of permissions that are applied to a sandbox are declared in a security policy file (represented at run-time by a `Policy` object). This policy file explicitly lists the permissions granted to a set (or multiple sets) of classes loaded from a particular location and/or cryptographically signed by a particular key. At run-time, the association between these sets of classes and their granted permissions is encapsulated within a `ProtectionDomain` object. The JRE provides a default security policy, which can be either supplemented or replaced by an administrator who wants to provide their own custom policy file when the JVM is started.

Some example vulnerabilities that affect the security manager include CVE-2008-5353 [19], CVE-2012-0507 [20] and CVE-2013-0422 [22]. Although these vulnerabilities are quite different (ranging from deserialisation issues to insufficient package access checking), the malware that targets them shares the same goal—to manipulate or disable the security manager so that arbitrary code can be executed. Exploits which target these types of vulnerabilities are what JMD is designed to detect.

3 Design and Implementation

In this section we outline our design choices and JMD's implementation details.

3.1 Assumptions

We have designed and implemented JMD to detect malware that successfully exploits vulnerabilities in Java's access control mechanisms (which operate in the second layer of Figure 2). In particular, we target malware that escapes the Java security sandbox by disabling or subverting the run-time security manager (as discussed in §2.2.2). Malware that targets vulnerabilities in either the Java class verifier (as discussed in §2.2.1) or in the Java native layer (as discussed in §2.1) are considered outside JMD's scope.

To put this in perspective, Gorenc et al.'s survey of Java vulnerabilities found that approximately half of the vulnerabilities patched between 2011 and 2013 had the ability to "bypass the sandbox and execute arbitrary code on the host machine" [8]. The top two vulnerability sub-categories in their sample set were "unsafe reflection" and "least privilege violation", both of which relate solely to Java's access control APIs and are thus potentially detectable by JMD. While the amount of malware targeting the Java native layer may be increasing [31], Java's access control model remains a popular target for malware authors—such exploits provide attackers with a "write once, run anywhere" weapon that does not require further customisation for a specific platform and/or OS [8].

3.2 Overview

Figure 3 provides a high-level overview of JMD's different stages.

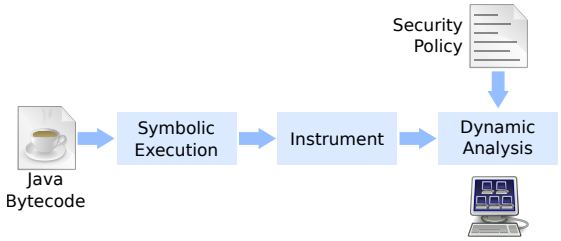

Figure 3: An overview of JMD

JMD takes a compiled Java application as input (Java bytecode, either as a `.class` or `.jar` file). In most cases this will be a Java applet, although JMD is able to analyse any Java application provided that it either (a) extends `java.applet.Applet` or (b) contains a static `main` method.

By default, a sample passes consecutively through JMD's three main stages of symbolic execution, instrumentation and dynamic analysis before reporting results to the user. However, JMD can optionally restrict its processing to certain stages, such as symbolic execution only, or instrumentation/dynamic analysis only.

The symbolic execution stage (see §3.3) attempts to determine what code paths exist within the sample, and to derive the trigger conditions that are required to execute them.

In the instrumentation stage, JMD injects custom logging code into the sample by using aspect-oriented programming (AOP) techniques (see §3.4). This logging code is designed to report access control violations, such as subversion of the security manager or unauthorised access to privileged APIs.

In the dynamic analysis stage (see §3.5), instrumented bytecode is executed within a virtual environment, in the context of a specific security policy. The instrumented bytecode detects and records any successful actions which require permissions that were not granted in the security policy. For example, if a sample is able to set the `SecurityManager` object to `null`—an operation that requires explicit permissions which are not given in JMD's default security policy— then JMD can conclusively infer that an access control vulnerability in the JRE has been successfully exploited. Note that dynamic analysis may be run multiple times, depending on how many trigger conditions are found during the symbolic execution stage.

At the conclusion of JMD's processing, results are returned to the user in an XML report.

The three stages are discussed in greater detail in the following sections.

3.3 Symbolic Execution

Symbolic execution is a mechanism for deriving the conditions required for a particular code path to execute [11]. For performing symbolic execution of Java bytecode, we use Symbolic PathFinder (SPF), which is an extension of the open-source Java PathFinder platform. In this section we focus on the differences between applying SPF to malware analysis and SPF's standard usage. A more detailed description of SPF's internals is available in [25].

SPF was primarily designed as a bug checking and test generation tool for Java applications. At a high level, SPF proceeds by replacing method arguments with symbolic values. Symbolic expressions are dynamically constructed and recorded as these symbolic values propagate through a given execution path.

These symbolic values and expressions are then used to derive the constraints required for a particular code path to be executed. Through a combination of searching and backtracking, SPF attempts to build constraints for all code paths to ensure maximum code coverage. Once SPF has finished processing the sample, a constraint solver attempts to determine the exact conditions required for each particular code path to be executed.

There are a number of technical issues that arise when applying SPF to the identification of trigger conditions in Java malware.[5] For example, SPF can not process bytecode unless it contains local variable debug information. A malware author would not normally include this information in the bytecode, as removing it increases the difficulty of reverse engineering and determining the malware's intention. Therefore, this debug information must be synthesised and injected into the bytecode before symbolic execution in SPF can occur.

SPF also requires the Java application to have a static `main` method as an entry point. However, Java applets do not have a `main` method—they rely on alternate entry points being invoked by the JVM at run-time.

Additionally, SPF only allows method arguments to be treated symbolically. However, it is common for malware to query the external environment and to take a specific code path that depends on the result of this query. Examples of external environment queries are given in Table 1. Given that SPF does not provide a mechanism to treat the result of these method calls symbolically, modifications were required to enable the exploration of code paths (and hence the identification of trigger conditions) that depend on their results.

The following sections describe how we extended SPF's behaviour in order to remediate these issues for JMD.

3.3.1 Symbolic Execution of Applets

As mentioned in §3.3, SPF only allows the execution of applications with a static `main` method as its entry point. This prevents the symbolic execution of Java applets, which require a browser or applet viewer to use the `init` or `start` methods as an entry point.

To overcome this restriction, a class containing a `main` method (based on Kurniawan's `AppletRunner` class [12]) was implemented for use as an entry point to run the applet within SPF.

We also implemented both a *model* and a *native peer* class to model the standard `Applet` class in SPF.[6] This enabled us to abstract away the underlying behaviour of a Java applet (which depends on functionality in the Java native layer; an element beyond SPF's symbolic execution capabilities). While our model and native peer classes were mostly sufficient for modelling malicious Java applets (which typically contain an exploit without any graphical component), this approach would be insufficient in modelling an applet containing an exploit that depended on the graphical capabilities available in the Java API.

3.3.2 Taint-based Symbolic Execution

We have used concepts from dynamic taint analysis [18] to allow SPF to construct constraints and explore code paths that depend on the external environment. Traditional dynamic taint analysis is the process of marking (or tainting) data that originates from an external (and possibly untrusted) source and tracking that data during application execution. However, dynamic taint analysis can also be combined with symbolic execution to construct constraints representing only the parts of execution that depend upon the tainted values [29]. Examples of external environment queries for which we want to construct constraints are listed in Table 1.

SPF provides a number of symbolic *listeners* that "gather and display information about the path conditions generated during the symbolic execution" [25]. We extended one of these listeners (the `SymbolicSequenceListener`) to allow local variables that store tainted data to be treated symbolically (in addition to method arguments, which are already treated symbolically by SPF).

Rather than blindly marking all local variables as symbolic in SPF (which would quickly lead to path explosion, discussed further in §4.4), JMD includes an SPF configuration option that allows the user to specify the signature of each method call that they wish to treat as producing tainted data. The tainted data (we limit ourselves to `String` objects) returned by these method calls is then marked as symbolic. SPF then treats this tainted data the same way it treats a symbolic method argument; i.e. constraints are constructed and solved in order to determine the trigger conditions required for a specific code path to be executed.

For example, the code in Figure 4 comes from a decompiled and deobfuscated malware sample. Using our taint-based symbolic execution approach, SPF can determine that a different code path will execute depending on the value returned by `System.getProperty("os.name")` (i.e. the OS running SPF). This information can then be provided to JMD's dynamic analysis stage, in which the (instrumented) malware would be executed four times, once for each result returned by `System.getProperty`.

3.4 Instrumentation

After SPF has completed symbolic execution of the Java malware, it can be instrumented and prepared for dynamic analysis.

The malware is instrumented using AspectJ, which is an open-source AOP extension to the Java language [13, 10]. AOP is a programming paradigm which seeks to achieve "separation of cross-cutting concerns" [10], where cross-cutting concerns are software components that impact (cut across) multiple

[5] A few examples of the issues we faced are mentioned here, but an exhaustive discussion is not included due to space constraints.

[6] Further information on model and native peer classes is available in [25].

Table 1: Examples of external environment queries

Method call	Description
`Applet.getParameter`	Customise an applet's operation via a name/value pair
`System.getProperty`	Query a system property
`System.getenv`	Query an environment variable

```
String s1 = System.getProperty("os.name").
    toLowerCase();
if (s1.indexOf("win") >= 0) {
    ...
} else if (s1.indexOf("mac") >= 0) {
    ...
} else if (s1.indexOf("nix") >= 0 || s1.
    indexOf("nux") >= 0) {
    ...
} else {
    return;
}
```

Figure 4: Decompiled and deobfuscated malware that alters its behaviour based on a tainted local variable

logical modules. The classic example of a cross-cutting concern is a logging library, which is usually implemented by linking a logger multiple times into several separate program modules. This results in increased levels of dependency and complexity, as the logging implementation becomes entangled with the core functionality provided by the program. By contrast, an AOP approach would maintain separation between a program's core logic and the logging components during the software development process. The components are subsequently combined together at compile or load-time to create the final system—a process known as *weaving*.

For JMD's purposes, we have exploited AspectJ's ability to weave existing bytecode (at both compile and load-time) in order to monitor Java's access control mechanisms. Figure 5 shows the important parts of the code which JMD weaves into its samples, with some of the finer details abstracted away or omitted. The `SecurityMonitor` aspect (i.e. the module expressing the cross-cutting concern, which is the monitoring of access control mechanisms) includes a pointcut[7] `anyCallOrExec` (line 11). This pointcut selects all method calls and their execution as join points (i.e. locations where the instrumentation code is weaved) to ensure malicious activities are recorded as early as possible.

The advice (lines 13–16) is executed whenever the `anyCallOrExec` pointcut picks out a join point (i.e. after every method call returns control to the caller, or a method body completes). This advice calls two methods: `checkSecMan` and `checkForIllegal Perms`. However, the advice is extensible and additional checks can easily be added.

The `checkSecMan` method (lines 18–26) compares the security manager's current state with its initial state (recorded on line 3). If the check returns an inconsistent result, an alert can be logged that indicates a compromise of the security manager.

`checkForIllegalPerms` (lines 28–40) ensures adherance to the permissions granted in the security policy file. The permissions for the currently-executing object are retrieved from the join point's context (lines 29–31). These permissions are then compared with the initial permissions specified in the policy file (line 5). If permissions that were not originally granted to the application are present, an alert can be logged indicating an access control compromise.

These methods (`checkSecMan` and `checkFor IllegalPerms`) attempt to verify the integrity of the Java security manager and its related classes. As discussed in §2.2.2, the ultimate goal of Java malware that seeks to subvert the Java sandbox is to disable (or change the state of) the security manager so that arbitrary code can be executed. By using instrumentation as described here to monitor the security manager and the set of permissions to which an application should have access, malware which successfully exploits a vulnerability in a Java access control API can be detected during JMD's dynamic analysis stage.

3.5 Dynamic Analysis

After instrumentation takes place, the sample is ready for dynamic analysis. To facilitate this, JMD includes support for two major virtualisation platforms: VirtualBox and VMware. A Cuckoo Sandbox [5] plugin was also developed.

Before dynamic analysis can occur, a sandbox virtual machine (VM) must be appropriately configured—i.e. an OS and a JRE must be installed. Given JMD's detection strategy of observing a sample's behaviour within a controlled environment, it is important that the JRE used during dynamic analysis be susceptible to the vulnerabilities targeted by the sample. As such, a sandbox VM can be configured with multiple JREs installed. The desired JRE for each JMD invocation can then be specified in a configuration file.

In order to execute Java applets in the sandbox VM, an appropriate HTML container file must first be generated. This HTML file includes any applet parameters that were found during the symbolic execution stage (see §3.3) or that have been explicitly entered by the user. During application execution (which involves executing the HTML file in a web browser or the JDK's `appletviewer` utility), the `SecurityMonitor` aspect detects and logs malicious activity targeting access control mechanisms as described in §3.4. If multiple combinations of applet parameters are found in the symbolic execution stage, then the applet is executed once for each combination. Once completed, JMD's logs are retrieved from the sandbox VM, the VM is (optionally) reverted to a clean snapshot and the logs are processed.

JMD produces an XML report describing its results. An example XML report is given in Figure 6. The report contains information on the runtime environment (gathered in the `SecurityMonitor`'s constructor) and the specific classes that performed any (detected) malicious actions. The example in Figure 6 shows that both an unauthorised permission and the disabling of the security manager were detected

[7] In AOP parlance, a *pointcut* is a program element which selects a particular *join point*—for example, a particular method signature like `System.out.println(String)` is a potential join point that could be expressed in a pointcut. Data from the execution context of this join point can then be queried and manipulated by third-party code within an *advice* block. For further details relating to AspectJ's implementation of pointcuts, advice and join points, see [10, 13].

```
1  public aspect SecurityMonitor {
2
3    private final SecurityManager initSecMan = System.getSecurityManager();
4
5    private final PermissionCollection initPerms = Policy.getPolicy().getPermissions(SecurityMonitor.class.
         getProtectionDomain());
6
7    public SecurityMonitor() {
8      // Gather information on the runtime environment
9    }
10
11   pointcut anyCallOrExec(): call(* *.*(..)) && execution(* *.*(..));
12
13   after(): anyCallOrExec() && !within(SecurityMonitor) {
14     checkSecMan();
15     checkForIllegalPerms(thisJoinPoint);
16   }
17
18   private void checkSecMan() {
19     SecurityManager secMan = System.getSecurityManager();
20
21     if (secMan == null) {
22       // Security manager disabled
23     } else if (!secMan.equals(this.initSecMan)) {
24       // Security manager altered
25     }
26   }
27
28   private void checkForIllegalPerms(JoinPoint jp) {
29     Class<?> clazz = jp.getStaticPart().getSourceLocation().getWithinType();
30     PermissionCollection pc = clazz.getProtectionDomain().getPermissions();
31     Enumeration<Permission> perms = pc.elements();
32
33     while (perms.hasMoreElements()) {
34       Permission p = perms.nextElement();
35
36       if (!this.initPerms.implies(p)) {
37         // Incorrect permission
38       }
39     }
40   }
41
42 }
```

Figure 5: AspectJ instrumentation code

by JMD. The XML report also gives a malicious/benign classification based on the behaviour detected by the `SecurityMonitor` (in this case it is malicious, due to the Java sandbox escape).

4 Evaluation

This section outlines our evaluation of JMD. We specifically focus on determining JMD's accuracy in detecting malware that targets Java's access control mechanisms. A discussion on possible evasion strategies is also provided.

4.1 Test Methodology

To test JMD's accuracy in detecting Java malware we collected a range of malware samples (targeting Java versions 6 and 7) from a variety of publicly-available sources. These sources included: Metasploit modules[8]; the Contagio blog[9]; and other computer security blogs. This resulted in an initial sample set of 228 samples.

Each sample was uploaded to VirusTotal[10] to determine the CVE number of the vulnerability targeted by the exploit. Where multiple VirusTotal scanners returned different results, a manual investigation (e.g. an Internet search for the hash and/or manual inspection of the sample) was used to determine the most probable CVE the sample represented.

From these results, the sample set was curated to remove broken samples and those that targeted out-of-scope elements (as discussed in §3.1—e.g. the class verifier and the Java native layer). The remaining 91 samples (which encompassed fourteen of the JRE access control-related CVEs disclosed over the past four years[11]) were labelled with the vulnerable JRE version(s)— i.e. JRE 1.6u0 and/or JRE 1.7u0. Java 8 was not evaluated due to its relative immaturity.[12]

By testing JMD against the earliest releases of Java 6 and 7, we ensured that (where our technique was successful) JMD would detect exploits for which patches were released partway through the Java version's lifecycle. The final list of samples is shown in Table 2.

These 91 samples were used as input to JMD's instrumentation and dynamic analysis stages (see §4.3 for a discussion of our attempts to use the symbolic execution stage in our evaluation). Dynamic analysis was performed in both a stand-alone Ubuntu 12.04 32-bit sandbox VM and in Cuckoo Sandbox using a Windows XP 32-bit sandbox VM. The results of these analyses were collated and are the topic of discussion in the following section.

[8] http://www.metasploit.com
[9] http://contagiodump.blogspot.com
[10] http://www.virustotal.com

[11] We attempted to quantify the total number of JRE CVEs from this period related to access control vulnerabilities (as opposed to vulnerability classes that are out-of-scope for JMD). However, we could not find public information sources that exhaustively correlated JRE CVEs with their vulnerability classes.

[12] At the time of writing, Java 7 is still offered as the default download at https://www.java.com.

```
<?xml version="1.0"?>
<jmdresults>
  <jmdresult targetFile="/home/metasploit/java/multi/browser/java_jre17_jmxbean/VaZbkzEM.jar">
    <result>malicious</result>
    ...
    <notes>
      <note>Sandbox Java Runtime: Java (TM) SE Runtime Environment</note>
      <note>Sandbox Java Runtime Version: 1.7.0-b147</note>
      <note>Sandbox Java VM: Java HotSpot(TM) 64-Bit Server VM</note>
      <note>Sandbox Java VM Version: 21.0-b17</note>
      <note>Sandbox OS Type: Linux</note>
    </notes>
    <classes>
      <class classname="B">
        <metadata>
          <filename>B.class</filename>
          <sourcefilename>B.java</sourcefilename>
          <classformatversion>46.0</classformatversion>
          <accessflags>ACC_PUBLIC|ACC_SUPER</accessflags>
        <metadata>
        <maliciouselements>
          <maliciouselement>Unauthorised use of java.security.AllPermission &lt;all permissions&gtl &lt;all
              actions&gt;</maliciouselement>
          <maliciouselement>Unauthorised manipulation of SecurityManager: security manager has been disabled
              </maliciouselement>
        <maliciouselements>
      </class>
      ...
    </classes>
  </jmdresult>
</jmdresults>
```

Figure 6: Sample XML report produced by JMD

Table 2: JMD evaluation sample set

CVE	# samples	Vulnerable JRE 1.6u0	Vulnerable JRE 1.7u0
CVE-2012-0507	40	✓	✓
CVE-2013-0422	17		✓
CVE-2008-5353	8	✓	
CVE-2013-0431	6		✓
CVE-2011-3544	5	✓	✓
CVE-2010-0840	5	✓	
CVE-2010-0094	2	✓	
CVE-2012-4681	2	✓	✓
CVE-2012-5076	2		✓
CVE-2012-5088	1		✓
CVE-2013-1488	1		✓
CVE-2013-2423	1		✓
CVE-2013-2460	1		✓
CVE-2013-2465	1	✓	✓
Total	92		

4.2 Experimental Results and Discussion

Tables 3 and 4 show the number and percentage of samples correctly identified as malicious by JMD. As JMD detects exploits within the JVM layer (which is independent of the underlying OS), we observed identical results in both sandbox environments. It is important to note that although Table 2 lists both JREs as being vulnerable to CVE-2012-4681, the samples we had for this CVE were compiled with Java 7 and hence were unable to run in Java 6. Similarly, both JREs are vulnerable to CVE-2013-2465, but the particular sample we had for this CVE (from Metasploit) was only confirmed to work in Java 7.

JMD was able to achieve a 100% success rate for ten of the fourteen CVEs in our sample set (across both JREs tested). However, some samples remained incorrectly classified by JMD. We subsequently examined these samples in greater detail.

CVE-2010-0094. The incorrectly classified CVE-2010-0094 sample was unique in our sample set in that it provides the user with an interactive GUI (via a text field and submission button). All of our other samples launch their exploit directly from the applet's `init` method (without requiring user interaction). Because this sample requires user interaction, JMD was unable to reach the exploitation and payload stages. Symbolic execution was also unable to assist because SPF does not model graphical elements (as discussed in §3.3.1).

Additionally, JMD was unable to instrument a particular method in this undetected CVE-2010-0094 sample. This was because the method was already approaching the maximum allowable method size, and the extra AspectJ instrumentation code put it over the limit[13]—the ramifications of this are discussed further in §4.4.

CVE-2011-3544. This CVE relates to vulnerabilities in the JRE's JavaScript engine. Half of the CVE-2011-3544 samples were incorrectly classified because they execute both their exploit and payload in the JavaScript engine. This differs from the detected samples, which only execute their exploit (disabling of the security manager) in the JavaScipt engine before returning execution to the applet to deliver its payload (which JMD's instrumentation

Table 3: Detection results for Java 6. This includes the number and percentage of samples successfully detected as malicious by JMD

CVE	# detected	% detected
CVE-2012-0507	37	93
CVE-2008-5353	8	100
CVE-2011-3544	3	60
CVE-2010-0840	5	100
CVE-2010-0094	1	50
Total	54	90.00

[13] Java methods are limited to a maximum of 64kB by the JVM specification.

Table 4: Detection results for Java 7. This includes the number and percentage of samples successfully detected as malicious by JMD

CVE	# detected	% detected
CVE-2012-0507	37	93
CVE-2013-0422	15	88
CVE-2013-0431	6	100
CVE-2011-3544	3	60
CVE-2012-4681	2	100
CVE-2012-5076	2	100
CVE-2012-5088	1	100
CVE-2013-1488	1	100
CVE-2013-2423	1	100
CVE-2013-2460	1	100
CVE-2013-2465	1	100
Total	70	90.91

correctly detects). For the `SecurityMonitor` to detect the incorrectly classified samples, the JRE's JavaScipt engine must be instrumented in the same way the sample is—this is beyond JMD's scope.

CVE-2012-0507. Our initial testing found that JMD could only detect nineteen (48%) of the CVE-2012-0507 samples. After some investigation, we found that this was due to many of the CVE-2012-0507 samples embedding a payload class file inside another class file as a byte array. This byte array undergoes an XOR deobfuscation routine (thus hiding the byte array's true intent from cursory static analysis) before being passed as an argument to `ClassLoader.defineClass`. This embedded payload class is typically defined under an unrestricted protection domain (i.e. a `Protection Domain` with an `AllPermission` permissions object), thus providing the payload with access to sensitive resources and operations (e.g. write to the file system, open a network connection, etc).

This privilege escalation was not detected by the `SecurityMonitor`'s `checkForIllegal Perms` method (as discussed in §3.4) because the obfuscated inner payload classes were not being weaved with our instrumentation code. In our initial implementation, we were only weaving precompiled bytecode (i.e. `.class` files); classes that were obfuscated and encoded in byte arrays (or by other means) and loaded at run-time were not instrumented. As a result, JMD was not able to log access control violations that were implemented in these inner payloads.

To rectify this issue, we added AspectJ's load-time weaving [13] functionality to JMD in order to instrument classes as they are loaded by the JVM's class loader. This increased our detection rate for CVE-2012-0507 samples from 48% to 93%.

4.3 Symbolic Execution Discussion

Unfortunately, JMD's symbolic execution stage did not produce the impact we had hoped it would. This was due to a number of factors.

Firstly, the taint-based approach proposed in §3.3.2 focuses on `String` objects as tainted data. Therefore, we rely heavily on SPF's ability to solve string-based constraints. While much work has been done to incorporate string-based constraints into SPF [26], the functionality required for SPF to handle many string methods remains unimplemented. For example, we encountered malware that called the `to LowerCase`, `toCharArray` and `split` string methods. We were able to implement symbolic handling for some of these methods (such as `toLowerCase`) ourselves, but others remain unimplemented due to time constraints. We also frequently encountered malware that converted `String` objects to arrays (and vice versa). Unfortunately, we found SPF's handling of this conversion and support for arrays relatively immature, resulting in unsolved constraints.

We also found that SPF had difficulties with the more heavily-obfuscated samples, especially those that made use of Java's reflection APIs (e.g. to instantiate classes, execute methods, etc.). SPF was often unable to construct constraints in these cases, potentially leaving execution paths unexplored.

Additionally, we found very few samples within our evaluation sample set that attempted to hide their payload behind a trigger condition. While obfuscation to prevent static analysis was common (e.g. randomising method names, field names and strings), hiding malicious activity behind trigger conditions (such as those discussed in §3.3 and [2]) was not. Our sample set only included one sample that hid its behaviour behind a specific applet parameter.

More commonly, applet parameters were used to directly inject metadata (e.g. they contained a URL to connect to, a port to connect on, a specific file to download, etc.) and were not used to construct constraints (i.e. they were not used at branch points to drive execution down a particular code path). For the one sample that did hide its exploit behind an applet parameter, SPF was unable to solve its constraints and derive a valid applet parameter value. (In this particular case, the conversion of the applet parameter `String` value into an array type prevented SPF from solving the constraints.)

However, the symbolic execution stage did provide useful information on payload creation and customisation (e.g. how the malware determined the underlying OS, such as the example in Figure 4) and applet parameter names (although in most cases SPF was unable to solve the required constraints and derive the parameter values that would enable increased code coverage during dynamic analysis).

4.4 Possible Evasion Strategies

Denial-of-service (DoS) attacks are possible on each of JMD's stages. The first DoS attack considered here affects JMD's symbolic execution stage. Path (or state-space) explosion is a common problem for symbolic execution engines [6, 16], and occurs when the application contains a large number of branch points. As the number of branch points grows, the number of possible code paths increases exponentially. Each new code path necessitates an increase in computational resources (both in time and memory) in order to explore all paths and solve the constraints required to execute those paths. It is therefore possible for malware to hide an exploit from JMD's symbolic execution stage by embedding it within a complicated code path (e.g. consisting of a very high number of branch points) that only triggers under specific conditions. In such a case, symbolic execution becomes computationally infeasible, and the exploit's trigger conditions will not be derived. Thus, the exploit will not trigger during dynamic analysis and will remain undetected.

A DoS on JMD's instrumentation stage is also possible, by exploiting the specified size constraints imposed on a Java class file. For example, bytecode crafted with 64 kB methods (the maximum allowable method size [14]) cannot be weaved with additional code by JMD's instrumentation stage. As such, any

malicious activity performed by these methods may go undetected during dynamic analysis.

Finally, it is possible for malware to examine itself to determine if it has been instrumented. For example, the malware could calculate a hash of itself at run-time and compare it to a hash calculated at compile time (and stored in the bytecode). This comparison would return a different result (because the code introduced by AspectJ would alter the malware's hash), leading the malware to infer that tampering had occurred. Alternatively, the malware could use the Java reflection API to determine whether any AspectJ classes had been woven into the bytecode. After determining if it had been instrumented, the malware could lie dormant and hence remain undetected during dynamic analysis. (The application of symbolic execution to Java's reflection APIs could theoretically be used to defeat these evasion strategies.)

5 Related Work

While much work has been undertaken into malware detection in general, relatively little research has examined Java malware specifically.

The Jarhead tool developed by Schlumberger et al. [28] uses static analysis and supervised machine learning techniques to detect malicious Java applets. Features are extracted from the Java bytecode and supplied to a classifier that is able to classify a malicious applet based on a training set of known malicious and benign applets. These features range from code metrics (e.g. the number of instructions and the code's cyclomatic complexity) to behavioural features (e.g. extending the `ClassLoader` class and the use of methods that are able to write files). A disadvantage of using a supervised machine learning algorithm is that the training data may be subject to overfitting. This could prevent the detection of zero-day exploits where either the relevant features have not been collected or there are not enough training samples to represent the new exploit technique. Additionally, because feature extraction is performed statically, the malware author may use obfuscation to hide key features (although the presence of obfuscation itself was a key feature in their results).

In [33], Wang proposed a dynamic analysis tool that records calls to the core Java API during execution. To achieve this, the core Java API's source code is patched to record key method calls (e.g. `System.setSecurityManager`) and then recompiled. Rules are defined for specific CVEs so an exploit's API trace can be matched to a rule set and hence a CVE. However, while this detects known exploits it does little in detecting zero-day attacks. Some heuristics for zero-day detection are proposed, however they are not evaluated. Additionally, patching the core Java classes may be against the Java license agreement.[14]

Soman et al. [30] proposed a similar technique, in which security-related operations are logged as events that are supplied to a signature-based intrusion detection system. However, the key differentiator between this technique and Wang's is that in [30] the JVM is instrumented to log security-related events (as opposed to patching the Java API in [33]). This has the advantage that system calls and calls to native code can be logged and used to detect attacks outside of the JVM. However, [30] still relies on a signature-based mechanism for detecting attacks.

[14] Section F of the Oracle Binary Code License Agreement, `http://www.oracle.com/technetwork/java/javase/terms/license/`.

An alternate approach for containing (rather than detecting) malicious Java applets was proposed by Chiueh et al. in [3]. Chiueh et al. proposed Spout, a transparent proxy that attempts to confine the damage of a malicious Java applet to an untrusted, disposable host (usually outside a firewall) that does not store sensitive data and is not used for any critical purposes. Spout achieves this by separating the application logic from the GUI component—the former executing on the disposable host and the latter executing on the host requesting the applet. This confines the malicious application logic to the disposable host, minimising the effects of an exploit. A key assumption in this approach is that "only the application logic component can damage the host machine's system resources" [3]. However, the authors of [8] found that during the period from 2011 to 2013 the 2D and AWT subcomponents were the second and fifth most vulnerable subcomponents in the Java language respectively. This potentially leaves Spout open to exploits that target vulnerabilities in these subcomponents.

Symbolic execution for malware analysis has been a popular research area [1, 2]. However, this has typically focused on native x86-based malware; Java malware has received comparatively little attention. While the Java-based SPF has been used for a number of purposes (e.g. test input generation [25, 6]) to our knowledge it has not been used in the context of Java malware analysis and detection. Similarly, AOP techniques have also previously been applied to the security domain (e.g. to replace insecure function calls and log security-relevant data [32]). Once again, to our knowledge AOP has not been used in the context of Java malware detection.

6 Conclusions and Future Work

In this paper we have presented the Java Malware Detector (JMD), a hybrid approach that combines symbolic execution, instrumentation and dynamic analysis to detect malicious Java applications. We demonstrated that it is possible to implement such a system using a number of popular open-source software platforms—specifically Java Pathfinder and AspectJ. Our evaluation on real-world malware samples shows that JMD was able to successfully detect malware variants representing fourteen of the known access control-related CVEs disclosed over the past four years. While there are known limitations in JMD's detection capability, the design and extensibility of JMD's instrumentation stage provides a platform upon which our results can be built and improved.

The application of symbolic execution to derive trigger points did not yield much success, but this was largely due to the incomplete state of SPF's symbolic `String` handling (see §4.3) and other issues which could potentially be remediated by extra engineering effort on SPF.

While JMD is successful in detecting the subversion of Java's access control mechanisms, there are a number of potential avenues to explore that could further expand JMD's capabilities. These avenues include: remediating the evasion strategies discussed in §4.4; adding symbolic handling implementations for the full `String` API as well as other unimplemented types in SPF; exploring additional trigger points via symbolic analysis, such as date and time triggers and network communication (e.g. receiving commands from a botnet Command and Control server); and extending the dynamic analysis engine to include at-

tacks on the class verifier and Java native layer. Further evaluation of JMD on additional samples could also be performed.

References

[1] Brumley, D., Hartwig, C., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., Song, D. and Yin, H. [2007], Bitscope: Automatically dissecting malicious binaries, Technical report, In CMU-CS-07-133.

[2] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D. and Yin, H. [2008], Automatically identifying trigger-based behavior in malware, *in* W. Lee, C. Wang and D. Dagon, eds, 'Botnet Detection', Vol. 36 of *Advances in Information Security*, Springer US, pp. 65–88.

[3] Chiueh, T.-C., Sankaran, H. and Neogi, A. [2002], 'Spout: a transparent proxy for safe execution of java applets', *Selected Areas in Communications, IEEE Journal on* **20**(7), 1426–1433.

[4] Cisco [2014], 'Cisco 2014 annual security report'.

[5] *Cuckoo Sandbox* [2014], `http://www.cuckoosandbox.org`.

[6] Ghosh, I., Shafiei, N., Li, G. and Chiang, W.-F. [2013], Jst: An automatic test generation tool for industrial java applications with strings, *in* 'Proceedings of the 2013 International Conference on Software Engineering', ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 992–1001.

[7] Gong, L. [1997], 'Java security: present and near future', *Micro, IEEE* **17**(3), 14–19.

[8] Gorenc, B. and Spelman, J. [2013], 'Java everydays – exploiting software running on 3 billion devices', *Black Hat USA 2013* .

[9] Kaspersky [2013], 'Java under attack – the evolution of exploits in 2012-2013', *Kaspersky Lab Report* .

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. [2001], An overview of aspectj, *in* 'Proceedings of the 15th European Conference on Object-Oriented Programming', ECOOP '01, Springer-Verlag, London, UK, UK, pp. 327–353.

[11] King, J. C. [1976], 'Symbolic execution and program testing', *Commun. ACM* **19**(7), 385–394.

[12] Kurniawan, B. [2011], *Java 7: A Beginner's Tutorial*, 3 edn, Brainy Software.

[13] Laddad, R. [2009], *AspectJ in Action*, Manning Publications.

[14] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A. [2013], 'The java virtual machine specification – java se 7 edition', `http://docs.oracle.com/javase/specs/jvms/se7/html/`.

[15] McAfee Labs [2012], 'Combating malware and persistent threats', `https://blogs.mcafee.com/mcafee-labs/combating-malware-and-advanced-persistent-threats`.

[16] Moser, A., Kruegel, C. and Kirda, E. [2007*a*], Exploring multiple execution paths for malware analysis, *in* 'Proceedings of the 2007 IEEE Symposium on Security and Privacy', SP '07, IEEE Computer Society, Washington, DC, USA, pp. 231–245.

[17] Moser, A., Kruegel, C. and Kirda, E. [2007*b*], Limits of static analysis for malware detection, *in* 'Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual', pp. 421–430.

[18] Newsome, J. and Song, D. [2005], Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, Technical Report CMU-CS-04-140, Carnegie Mello University.

[19] NIST [2008], 'CVE-2008-5353', `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-5353`.

[20] NIST [2012*a*], 'CVE-2012-0507', `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0507`.

[21] NIST [2012*b*], 'CVE-2012-1723', `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-1723`.

[22] NIST [2013], 'CVE-2013-0422', `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0422`.

[23] Oracle Corporation [2014], 'Java security overview', `http://docs.oracle.com/javase/7/docs/technotes/guides/security/overview/jsoverview.html`.

[24] Oracle Corporation [n.d.], 'Java platform standard edition 7 documentation', `http://docs.oracle.com/javase/7/docs/index.html`.

[25] Păsăreanu, C., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P. and Rungta, N. [2013], 'Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis', *Automated Software Engineering* **20**(3), 391–425.

[26] Redelinghuys, G., Visser, W. and Geldenhuys, J. [2012], Symbolic execution of programs with strings, *in* 'Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference', SAICSIT '12, ACM, New York, NY, USA, pp. 139–148.

[27] Schierl, M. [n.d.], 'CVE-2012-1723 – oracle java applet field bytecode verifier cache remote code execution [cve-2012-1723 openjdk: insufficient field accessibility checks (hotspot, 7152811)]', `http://schierlm.users.sourceforge.net/CVE-2012-1723.html`.

[28] Schlumberger, J., Kruegel, C. and Vigna, G. [2012], Jarhead analysis and detection of malicious java applets, *in* 'Proceedings of the 28th Annual Computer Security Applications Conference', ACSAC '12, ACM, New York, NY, USA, pp. 249–257.

[29] Schwartz, E. J., Avgerinos, T. and Brumley, D. [2010], All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), *in*

'Proceedings of the 2010 IEEE Symposium on Security and Privacy', SP '10, IEEE Computer Society, Washington, DC, USA, pp. 317–331.

[30] Soman, S., Krintz, C. and Vigna, G. [2003], Detecting malicious java code using virtual machine auditing, *in* 'Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12', SSYM'03, USENIX Association, Berkeley, CA, USA, pp. 11–11.

[31] Tang, J. [2013], 'Java native exploits going up', http://blog.trendmicro.com/trendlabs-security-intelligence/java-native-layer-exploits-going-up/.

[32] Viega, J., Bloch, J. and Chandra, P. [2001], 'Applying aspect-oriented programming to security', *Cutter IT Journal* **14**(2), 31–39.

[33] Wang, X. [2013], 'An automatic analysis and detection tool for java exploits', *Virus Bulletin* .

[34] Zovi, D. A. D. [2011], 'Attacker math 101', https://www.trailofbits.com/resources/attacker_math_101_slides.pdf.