securityandtrust.lu

MUSTI: Dynamic Prevention of Invalid Object Initialization Attacks

Alexandre Bartel, Jacques Klein, Yves Le Traon

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

TR-SNT-2018-3

ISBN: 978-99959-58-05-3

June 15, 2018

Version 1.0

UNIVERSITÉ DU LUXEMBOURG

MUSTI: Dynamic Prevention of Invalid Object Initialization Attacks

Alexandre Bartel
University of Luxembourg, SnT

Jacques Klein
University of Luxembourg, SnT

Yves Le Traon
University of Luxembourg, SnT

Abstract

The invalid object initialization vulnerability has been known at least since the 1990's when discovered by a research group at Princeton University. Recently, such a vulnerability, identified as CVE-2017-3289, was found again in the bytecode validation code of the Java virtual machine. In this paper, we explain what the vulnerability is and how it could be used to bypass Java sandbox restrictions, leading to a situation in which the security of the Java virtual machine is compromised. We then present a solution called MUSTI to detect and prevent attacks leveraging this kind of critical vulnerability at runtime. MUSTI, has been evaluated on the Dacapo Java benchmark as well as on real-world programs and has a runtime overhead below 5%. We also show how MUSTI can be optimized to have a runtime overhead below 1%.

1 Introduction

When the Java language was introduced in the mid-1990's, it was thought that the language is more secure than C/C++ because it does not allow to directly manipulate the memory – it uses a garbage collector instead – and because array bounds are automatically checked at runtime. This design makes certain vulnerabilities such as buffer overflows a thing of the past. Unfortunately, the Java Virtual Machine (JVM) and part of the Java library are still written in C/C++ code which makes the whole Java architecture still vulnerable to low level attacks.

Another Java feature, emphasized by Sun Microsystems at the time, is the fact that the JVM can run untrusted code in a sandbox and give this untrusted code only limited or no privilege at all. This was particularly convenient in web browser which could execute such untrusted code in so called *Applet*s [2] with the least privileges. Alas, even though the security architecture seemed fine, numerous security vulnerabilities have been found in Java which enable, in most cases, a total sandbox escape. This means that if an attacker can redirect a user to a web page he controls, he can run malicious Java code on the user's browser to escape the sandbox and potentially run code with the privileges of the web browser. The situation was so alarming for Java and other plugins based on the NPAPI, that major companies developing web browsers such as Google [10] and Mozilla [11] decided to first disable them by default and to then remove them altogether.

Today, a typical computer user navigates the world wide web without executing any Java code within his browser. However, it may still be the case that companies who rely on legacy software require their employees to activate the Java plugin in their browsers to access specific services. Also, some computer users – not necessarily within a company – may also choose to re-enable Java to access a particular service. Forcing the browser to use Java increases the attack surface and thus puts users at risk. As we will see in this paper, *invalid object initialization* vulnerabilities allow to bypass the Java sandbox. Our approach aims at improving the Java virtual machine to prevent such vulnerabilities at runtime.

Meanwhile, Oracle [1] did a lot of effort to improve the security of the Java platform. One approach they used is to reduce the attack surface. Indeed, the Java Class Library (JCL) has numerous legacy classes which are prone to contain security vulnerabilities. By marking them as "restricted" an attacker cannot directly instantiate them anymore and thus cannot use code that might have been useful to perform his attack. This effectively breaks existing exploits relying on such classes but also makes it harder for the attacker to find new code he can leverage in a new attack.

Oracle also developed an approach to automate the process of verifying the code and finding new security vulnerability [14]. The approach taints untrusted user data and checks if it flows to security sensitive operations such as the loading of a class in a privileged context. If it

[1] Oracle completed the acquisition of Sun Microsystems in 2010

does, the approach makes sure that objects created by the security sensitive operation do not flow back to the user context. This prevents, in our example, an attacker from using a class loaded by a security sensitive operation.

Unfortunately, despite these efforts, new vulnerabilities have been found. One the latest vulnerabilities, CVE-2017-3289 – an invalid object initialization vulnerability – is studied in this paper. We understand how the vulnerability works and devise a solution to prevent attacks at runtime. The contributions we make are the following:

- We analyze a recent Java vulnerability and a develop proof-of-concept for it
- We analyze the root causes leading to invalid object initialization vulnerabilities
- We propose an open-source solution, MUSTI, to prevent Java sandbox bypasses leveraging uninitialized instance vulnerabilities
- We evaluate MUSTI in terms of detection, runtime overhead and memory overhead

This paper is organized as follows. First, in Section 2, the Java security model is presented. Then, we explain what the invalid object initialization vulnerability is in Section 3. In Section 4 we present our approach to prevent the vulnerability at runtime. We evaluate our approach in Section 5. In Sections 6 and 7 we describe the limitations of the approach and discuss potential improvements. Related work is presented in Section 8. Finally, we conclude in Section 9.

System Classes (Trusted code)

```
 1 class ClassLoader {
 2   protected ClassLoader() {
 3     this(checkCreateClassLoader(),
            getSystemClassLoader());
 4   }
 5
 6   private static void checkCreateClassLoader() {
 7     SecurityManager security = System.
            getSecurityManager();
 8     if (security != null) {
 9       security.checkCreateClassLoader();
10     }
11     return null;
12   }
13 }
14
15 class SecurityManager {
16   public void checkCreateClassLoader() {
17     checkPermission(SecurityConstants.
            CREATE_CLASSLOADER_PERMISSION);
18   }
19 }
```

Application Classes (Untrusted code)

```
20 class UntrustedMain {
21   public static void main(String[] args) {
22     ClassLoader myCL = new ClassLoader() { };
23   }
24 }
```

Figure 1: Code without the `CREATE_CLASSLOADER` permission (UntrustedMain) cannot instantiate a class loader (line 22) because the security check (line 17) will throw a security exception.

2 The Java Security Model

In this section, we briefly present the fundamental concepts that are required to understand the Java security model: security policy, security domains, permissions, the security manager and the `doPrivilege` method.

2.1 Security Policy

Java code can be associated with a security policy. The policy is a list of permissions describing what the code is allowed to do. Usually, it makes sense to give more permission to trusted code and the least permissions possible to untrusted code. For instance, trusted server code can run with all permissions, while untrusted code running in the browser runs with no permission. To prevent the code from performing forbidden operations, Java runs untrusted code within a sandbox. For every sensitive operation the code tries to perform, the sandbox checks at runtime that the code is authorized. If it is not, a security exception is thrown. Untrusted code typically has no permission and, thus, cannot access the file system, the network, etc..

2.2 Security Domain

Every class in the JVM is loaded with a class loader and associated with a security domain. Classes shipped with the JRE (Java Runtime Environment) also known as *system* classes, are loaded with all permissions. An example of a system class is `java.lang.Class`. Untrusted classes downloaded from the Internet and running in an applet, or more generally all classes coming from an untrusted source, should be loaded with no permissions. Trusted classes can be loaded with all permissions but should be loaded with the least permissions to respect the principle of least privilege [24].

In this paper we suppose that the user is running untrusted code on the Java virtual machine. The untrusted code runs within the sandbox without any permission.

2.3 The Security Manager

Permissions are only checked when a security manager has been created and set. This can be done programmat-

SecurityManager.checkPermission()
SecurityManager.checkCreateClassLoader() (line 16)
ClassLoader.checkCreateClassLoader() (line 6)
ClassLoader() (line 2)
UntrustedMain.main() (line 21)

Figure 2: Call stack when the `checkPermission` method is called (Figure 1 line 17).

ically via a call to `System.setSecurityManager()` or with a command line option when launching the Java virtual machine. How the security manager is used when checking permissions is illustrated in Figure 1. In the constructor of the `ClassLoader` class there is a call to `checkCreateClassLoader()` (line 3). This method then calls `checkCreateClassLoader` of the security manager (line 9). Finally, the security manager calls `checkPermission()` to check for permission `CREATE_CLASSLOADER` (line 17). Notice that the security check is only performed if a security manager is set (lines 7-8). Thus, with a security manager set, untrusted code cannot instantiate a subclass of a `ClassLoader` since the constructor checks for the `CREATE_CLASSLOADER` permission.

2.4 Permission Checks

2.4.1 Normal Permission Check

When a permission check is performed, all elements of the stack trace (all methods that have been called since `main()`) are analyzed and must have the right permission. Otherwise, a `SecurityException` is thrown. When the code of Figure 1 is executed and reaches line 17, it has the call stack with five elements illustrated Figure 2. The `checkPermission` method goes backwards when analyzing the call stack. The three last methods `ClassLoader()`, `checkCreateClassLoader()` and `checkCreateClassLoader()` are from system classes and thus, have the right permission. Method `main()`, however, is from of an untrusted class, and does not have the `CREATE_CLASSLOADER` permission. Thus, `checkPermission` throws a `SecurityException`, which prevents the `ClassLoader` from being instantiated.

```
1  static boolean unaligned() {
2    if (unalignedKnown)
3      return unaligned;
4    String arch = AccessController.doPrivileged(
5      new sun.security.action.GetPropertyAction(
6        ”os.arch”)
7    );
8    unaligned = arch.equals(”i386”)
9      || arch.equals(”x86”)
10     || arch.equals(”amd64”)
11     || arch.equals(”x86_64”);
12   unalignedKnown = true;
13   return unaligned;
14 }
```

Figure 3: Example of `doPrivileged` call to read the `sys.arch` system property.

2.4.2 Do More with Less

Sometimes, code from the JCL (Java Class Library) has to execute $code_{priv}$, that may require more privilege than the current privileges of the running code which may be potentially untrusted code with no permission. To achieve that, Java comes with a built-in functionality called `doPrivileged()`. With the `doPrivileged()` all [2] permissions are temporarily given to the code to execute $code_{priv}$. An example of `doPrivileged()` is illustrated in Figure 3. The code at lines five and six requires a permission to get a system property. Without `doPrivileged`, this code would have thrown a security exception since the stack check would have hit the `main()` method which is potentially from an untrusted class with no permission. With `doPrivileged`, the stack check is still performed backwards but stops at the `doPrivileged` method. This enables untrusted code to execute code that temporarily require one or more permissions.

3 Uninitialized Instance Vulnerability

3.1 What it is

An uninitialized instance vulnerability enables the creation of an object which is not properly initialized. In the case of Java, this means that the chain of calls to constructors is broken resulting in some constructor methods not being called. The consequences are the following:

- code that should be executed may not be executed
- fields that should be initialized may not be initialized and may thus end up having “default” values (e.g., *null* for references)

Take the example of Figure 4 representing the class hierarchy of hypothetical Java library classes *A*, *B* and *C*. In this example, class *C* extends *A* and classes *A* and *B* both extend *Object*. In a normal program instantiating a new object of type *C*, the constructor of *C* starts executing. The first instruction of the constructor actually calls the constructor of *A* (Figure 5 line 11 right) which immediately calls the constructor of *Object* (Figure 5 line

[2] all or a subset of permissions

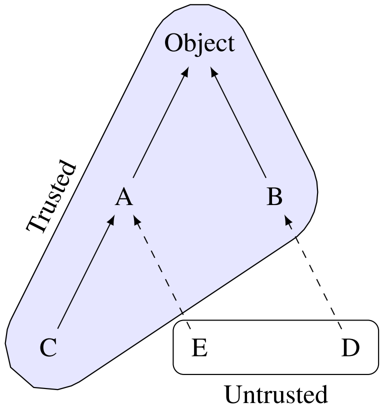

Figure 4: Class hierarchy for Java library classes *A*, *B* and *C* (representing trusted code) and application defined classes *D* and *E* (representing untrusted code).

10 left). When the constructor of *Object* terminates, the execution goes back in the constructor of *A* which continues and terminates. Finally, the constructor *C* continues and also terminates.

An uninitialized instance vulnerability allows to create an instance of an object whose constructor will not call the constructor of its super class (e.g., by exploiting a bug in the bytecode verifier). More concretely, if we have class *E* extending *A* and class *D* extending *B* (Figure 4), the vulnerability allows to create objects of type *E* without calling *A*'s constructor or objects of type *D* without calling *B*'s constructor.

3.2 Impact on Security

If access control or other security mechanisms rely on the value of fields initialized in constructors, an analyst could be able to bypass them by setting the field values to default values. Likewise, if security checks are directly performed in constructor code an analyst may be able to bypass them by not executing the code.

Figure 5 illustrates these two kinds of vulnerable code. The authorization check in the constructor is illustrated in the constructor of class *B*. If class *B* has a subclass such as *D*, controlled by the analyst, the constructor of *B* will not be called from *D* during an attack. The consequence is that the analyst can instantiate objects he should not be able to. Without the invalid object initialization vulnerability, the instantiation of such object would have thrown a security exception at runtime.

For the sake of completeness we also describe here the impact of uninitialized fields. The field used as condition for access control is illustrated in method *A.m* (lines 14-18, left). If class *A* has a subclass such as *E*, controlled by the analyst, the constructor of *E* will not call the constructor of *A*. The consequence is that field *forbidden* will have the default value of *false*. Any subsequent call to method *A.m* will succeed since the access control check is bypassed. Note that we assume it is bad practice and quite rare to use fields for access control. Finding such fields is thus out of the scope of this paper.

```
1   class Object {
2    <init>() {
3      ...
4    }
5   }
6
7   class A {
8    bool forbidden;
9    <init>() {
10    super();
11    forbidden = true;
12    ...
13   }
14   void m() {
15    if (forbidden) {
16     return;
17    }
18    ...
19  }
```

```
1   class B {
2    <init>() {
3     super();
4     checkPermission(”P1”);
5     ...
6    }
7   }
8
9   class C {
10   <init>() {
11    super();
12    ...
13   }
14  }
```

Figure 5: Constructor code for classes *Object*, *A*, *B* and *C*. Note that constructor *B* is checking for permission *P*1 and method *m* from class *A* is using a field initialized in a constructor for access control.

3.3 Vulnerability History

As far as we know, there are at least three publicly known *invalid object initialization* vulnerabilities for Java. The first publicly known invalid object initialization vulnerability has been found by a research group at Princeton in 1996 [16]. The vulnerability lies within the bytecode verifier. It allows for a constructor to catch exceptions thrown by a call to `super()` and return a partially initialized object. Note that at the time of this attack the class loader class did not have any instance variable. Thus, leveraging the vulnerability to instantiate a class loader gave a fully initialized class loader on which any method could be called. The second has been discovered by LSD [3] in 2002 [19]. The authors also exploited a vulnerability in the bytecode verifier which enables to not call the constructor of the super class. They have not been able to develop an exploit to completely escape the sandbox. They were able, however, to access the network and read and write files to the disk. The last one has been made public in 2017 and is CVE-2017-3289. This vulnerability is also a bug in the bytecode verifier and might allow an attacker to completely bypass the Java sandbox.

[3] a security research group called The Last Stage of Delirium

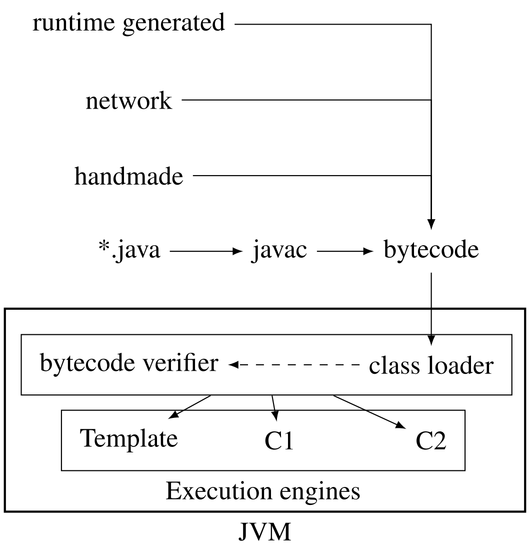

Figure 6: Simplified View of the Java Runtime: The JVM loads bytecode and may verify it before it is executed by one of the execution engines (*Template*, *C*1 or *C*2)

4 Preventing the Vulnerability

Our approach aims at preventing the exploitation of the vulnerability at runtime. We patch the Java virtual machine to add code which checks that objects have been correctly initialized, i.e. that the chain of constructors has not been broken. To understand where to patch the virtual machine we first have to understand how it loads and represents the code.

4.1 Code Loading in the JVM

As illustrated in Figure 6, the Java Virtual Machine (JVM) loads only bytecode. The bytecode however, can originate from multiple sources: compiled by the Java compiler (the usual), downloaded from the network, generated at runtime, or assembled manually (typical for exploiting a vulnerability in the bytecode verifier). The JVM loads the bytecode through a class loader. The bytecode is usually verified but not for classes which are deemed “trusted” such as classes in the packages “java.*”. At runtime, the bytecode of a Java method is either executed with the template engine, the C1 engine or the C2 engine. The C1 and C2 engines transform and optimize the bytecode and execute the resulting code. Which engine is chosen depends on the number of times the method has already been executed. The appropriate place to instrument the bytecode is thus within the class loader. Indeed, this is where the virtual machine loads a class and creates an internal representation of the class' bytecode. Since every bytecode that is loaded by the JVM has to go through the class loader, we instrument the bytecode there.

```
1  aload_0         1   aload_0
2  ...             2   aconst_1
3  return          3   putfield allow_in_constructor = 1
4  ...             4   aload_0
5  return          5   ...
                   6   goto new_label
                   7   ...
                   8   goto new_label
                   9   new_label:
                   10  aload_0
                   11  aconst_1
                   12  putfield bypass_check = 1
                   13  aload_0
                   14  aconst_0
                   15  putfield allow_in_constructor = 0
```

Figure 7: Constructor Transformation (pseudo assembly code). The original constructor code is shown on the left. Transformed or additional code has a gray background color.

4.2 Instrumenting Code in the JVM

Naively instrumenting existing bytecode may result in broken bytecode. Indeed, the instrumented bytecode must verify:

- that branching instruction offsets are still pointing to the right instruction,
- that try/catch blocks and handlers are still consistent
- that stack map frames [4] are appropriate and still consistent

Our approach is to add one field, *is_initialized* to the *Object* class and to modify the bytecode of the constructors of the *Object* class to initialize the field to *true*. When a method is called, it could thus first check that the field is correctly initialized. If it is, the method is executed normally. Otherwise, is means that the object on which the method is called has not been correctly initialized. The method does not execute and the program stops, for instance by throwing an exception.

Object Constructors Instrumentation Only constructors in `java.lang.Object` are instrumented. The modifications are presented in Figure 7. First, code is prepended to the constructor bytecode to set the field *allow_in_constructor* to *true* (lines 1 to 3). This allows the constructor to call methods even if it is not fully initialized yet. Then, every `return` instruction is changed to a `goto` instruction to branch to the appended

[4] structures to help type checking the bytecode [7]

```
1    aload_0
2    ...
3    return
4    ...
5    return
```

```
1    aload_0
2    invokevirtual isInit()
3    ifne new_label
4    new SecurityException
5    athrow
6    new_label:
7    aload_0
8    ...
9    return
10   ...
11   return
```

Figure 8: Method Transformation (pseudo assembly code). The original method code is shown on the left. Transformed or additional code has a gray background color.

code. The appended code (lines 9 to 15) sets back the field *allow_in_constructor* to *false* and sets the field *is_initialized* to *true* indicating that the object has been correctly initialized.

Method Instrumentation The bytecode transformation for methods is illustrated Figure 8. Code is prepended to the method bytecode to check if the constructor has been properly initialized (lines 1 and 2). If it is the case, the method is executed normally (lines 3 and 6). Otherwise, the method throws a security exception (lines 4 and 5).

4.3 Implementation of Musti

We use the Java virtual machine from OpenJDK 8 update 144 branch 01 (see Appendix A for more details).

We modify `method.cpp` to add code which modifies the bytecode of existing methods. We rely on code already present in the JVM file `relocator.cpp` to instrument the bytecode of methods. In theory, we modify `classFileParser.cpp` to add code to instrument the constructors of the `java.lang.Object` class. In practice, as explained in the next paragraph, we modify `classFileParser.cpp` to instrument all constructors of all classes extending `java.lang.Object`. Overall, the new code accounts for about 2000 lines of C++.

While implementing our solution we faced one major challenge which is that the `java.lang.Object` class cannot be easily modified. According to multiple sources [5] and our experience, adding a field or a method to this class would require heavy modification of the Java virtual machine source code since numerous parameters for this class are hardcoded throughout the source code of the virtual machine. We solved this by modifying all classes which immediately extend `java.lang.Object`.

5 Evaluation

In this section, we answer the following research questions:

- RQ1: can Musti prevent attacks based on unitialized instance vulnerabilities?
- RQ2: what is the runtime overhead of Musti?
- RQ3: what is the memory overhead?
- RQ4: how many constructors are vulnerable?

All the experiments ware performed on a machine with 32Gb of RAM and an Intel Core i7-6700HQ CPU @ 2.60GHz featuring 8 processors each having 8 cores.

5.1 RQ1: Preventing Attacks

The main goal of Musti is to prevent attacks based on invalid object initialization. We reverse engineered the patch of vulnerability CVE-2017-3289 [6] to create an exploit. This exploit features unprivileged code which leverages the vulnerability to create an instance of `java.lang.ClassLoader`. How the vulnerability has been reversed and the exploit created is detailed in Appendix B. Note that all publicly known invalid object initialization vulnerabilities are located within the code of the bytecode verifier.

The exploit has no permission and can nonetheless create an instance of `java.lang.ClassLoader` on a vulnerable version of the Java virtual machine. However, in our modified version Musti, the added bytecode in the `java.lang.ClassLoader` constructor detects that the constructor call chain has been broken since the field *is_initialized* is still set to *false*. It thus successfully stops the program before it can leverage the vulnerability.

Musti is able to successfully prevent attacks leveraging invalid object initialization vulnerabilities present within the bytecode verifier.

5.2 RQ2: Overhead of Musti

We evaluate Musti on DaCapo [13], a benchmark suite intended as a tool for Java benchmarking, as well as on two real world Java programs: Soot [22] and JavaML [12]. For every target test suite or program, we run the program 50 times in a row in the same Java virtual machine. We do this to be able to stabilize the running time of the Java virtual machine. Indeed, the JVM is quite a complex software which requires to load classes used in the target program and which comes with many optiminzations to improve the running time. In order to

[5] https://stackoverflow.com/add-a-field-to-java-lang-object https://stackoverflow.com/instrumenting-array-via-java-lang-object

[6] the vulnerability lies at the bytecode verifier level within the JVM

evaluate a target program in its best optimized version and to remove noise related to class loading and code optimization, we run it 50 times and use the last 10 runs as representative of the best optimized version of the program. Every run with 50 iterations is repeated 10 times. The final version uses the mean running time of every iteration.

5.2.1 DaCapo Benchmark

The DaCapo benchmark is not maintained anymore. Therefore, some of the benchmark's test suites are not working against Java 8. Thus, we only rely on a subset of all the available test suites, namely `luindex`, `lusearch`, `pmd` and `xalan`. The first two are based on Apache Lucene [7]: `luindex` uses lucene to indexes a set of text documents such as the works of Shakespeare, while `lusearch` uses lucene to do a text search of keywords over text documents. Regarding the two other test suites, `pmd` analyzes a set of Java classes to detect potential problems and `xalan` transforms Xml documents into Html documents.

We run all the test suites on both the original JVM, *orig*, and the modified JVM, MUSTI. We use three different versions of the modified JVM. In the first version, *naive*, all methods of all classes are instrumented. In the second version, $opti_1$, only methods of classes which are public and non-final are instrumented. In the third version, $opti_2$, only methods of classes checking for a permission in one of their constructors are instrumented. How the list of such constructors has been computed is the topic of RQ4 in Section 5.4.

The results are shown in Figure 9 to 12. Figure 9 represents the results of the experiments for `luindex`. The first graph compares the original JVM, *orig*, (crosses) with the naive modified JVM, *naive* (circles). The second graph compares *orig* with $opti_1$ while the third one compares *orig* with $opti_2$. The overhead is indicated in the three graphs with triangles. Figure 10 represents the result for `lusearch`, Figure 11 for `pmd` and Figure 12 for `xalan`.

The first observation is that the overhead (taken on the last 10 runs) decreases from *naive* to $opti_1$ to $opti_2$. For `xalan`, for instance, it goes from 5.17% to 4.44% to 2.82%. For `pmd` it goes from 11% to 8.06% to -0.35%. The last overhead is negative meaning that the modified JVM ran faster than the original one. The explanation is that since the run times of *orig* and *modif* are very close, it may happen that the average of *modif* is faster than the average of *orig*, hence the negative value close to 0%.

The second observation, is that for the first few runs, *modif* has a very high overhead compared to *orig*. This is explained by the fact that in the modified JVM, the bytecode of a huge number of methods is modified which takes time. This is one of the reason, we execute 50 runs of the same program in the same virtual machine to get rid of the class loading impact on the overall running time of the program. This phenomenon is specially interesting in the case of `xalan`. After iteration 18, the overhead goes from more than 150% down to the range 0-5%. This is not only explained by the method bytecode modification but also by the numerous optimizations the virtual machine performs on the bytecode. These modification are highly dependent on the number of execution of the method in question, i.e. the more a method is executed, the more the JVM tries to optimize its code. It is likely that for the case of `xalan` a huge batch of method is used frequently and ends up being highly optimized at the 18th iteration.

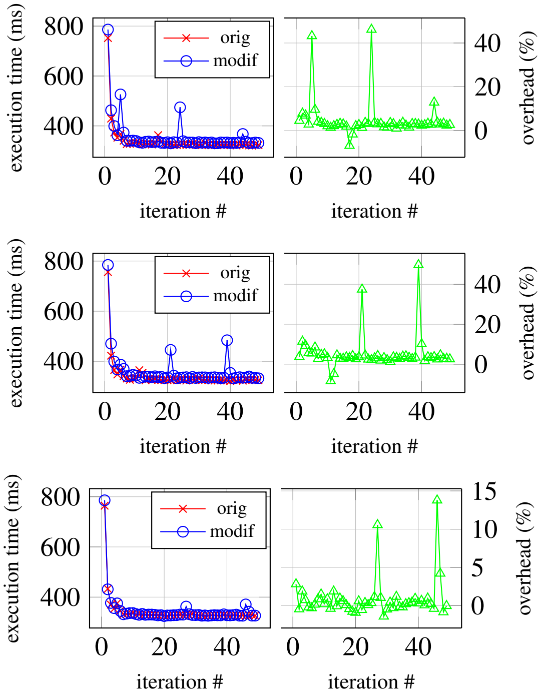

Figure 9: `luindex` naive (up), opti1 (middle) and opti2 (down)

The third observation is that the overhead of $opti_2$ is close to zero. This means that the modification of the JVM to detect invalid object initialization has almost no impact on the runtime of the program.

The fourth observation is that there are *bursts* of the overhead. This is especially noticeable in Figure 9. We assume this is caused by the garbage collector which takes time to remove all the unused objects and thus increases the running time for some iterations. The negative effect on the overhead can be observed both for the

[7] http://lucene.apache.org

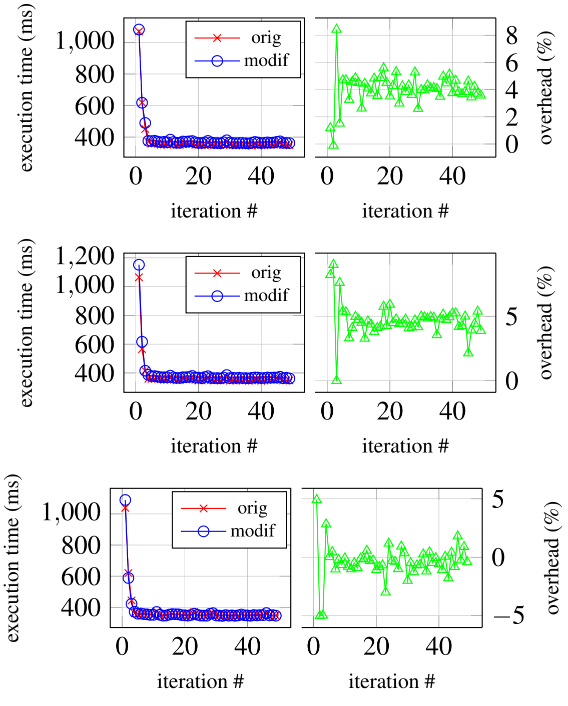

Figure 10: `lusearch` naive (up), opti1 (middle) and opti2 (down)

original VM and for MUSTI. Figure 14 illustrates this behavior. At iteration 17, the garbage collector runs for the original VM causing the overhead to be negative. At iteration 19, the garbage collector runs for MUSTI, causing the overhead to be way higher.

The evaluation on the DaCapo benchmark indicates that impact of MUSTI on the running time is low and is in the range 0-3%.

5.2.2 Real Java Software

We also evaluate MUSTI on two real Java programs: Soot [22] a program to analyze and optimize Java bytecode and Java-ML [12] a machine learning library. As for the DaCapo benchmark, we run each program 50 times in the same virtual machine to remove noise from class loading and code optimization. We run Soot to transform the 7Mib Dalvik bytecode of one Android application to the internal representation of Soot called Jimple and to output this representation to the file-system. We run Java-ML on the tutorial examples available in the source code: random forest, kmeans, store data, Weka classifier, ARFF loader, Weka clusterer, sampling, feature scoring, feature ranking, feature subset selection, ensemble feature selection, sparse instance, dataset, dense

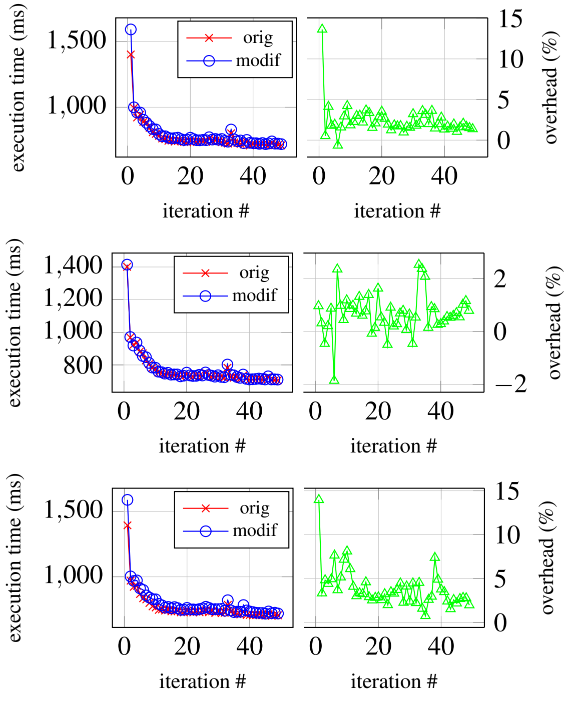

Figure 11: `pmd` naive (up), opti1 (middle) and opti2 (down)

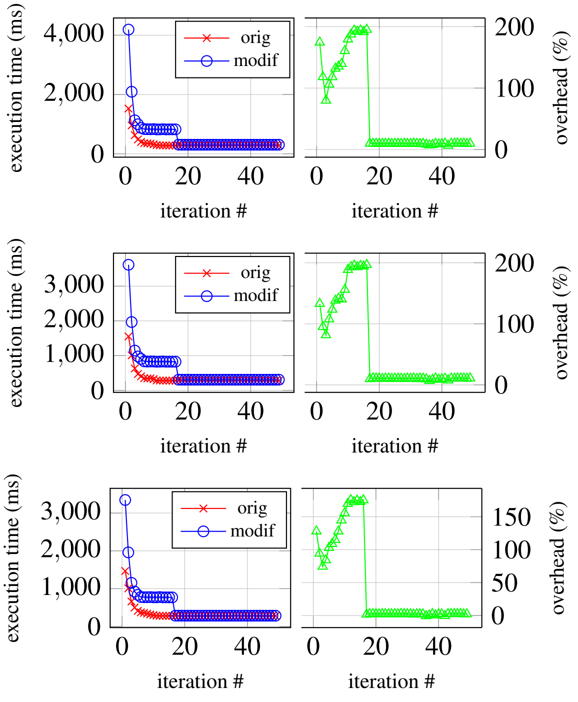

Figure 12: `xalan` naive (up), opti1 (middle) and opti2 (down)

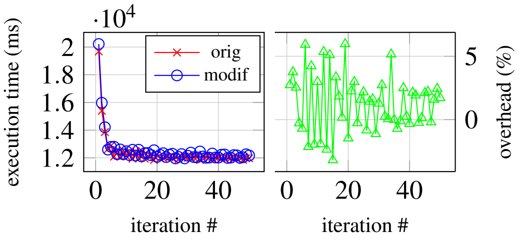

Figure 13: `Soot` *opti*2

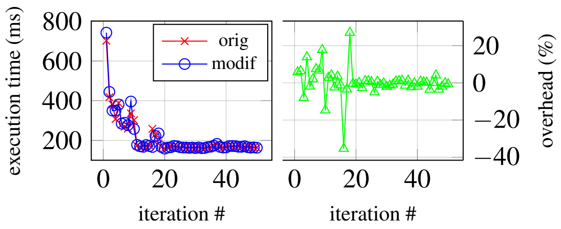

Figure 14: `JavaML` *opti*2

instance, lib SVM, self optimizing lib SVM, KNN, naive Bayes, cross validation, k-dependent Bayes and entropy partitioning. We only run Soot and Java-ML with $opti_2$, the version of the modified JVM which yielded the best results (the lower overheads) for DaCapo. The precise versions of the tools and the Android application are listed in Appendix A.

Figure 13 represents the results for Soot and Figure 14 for Java-ML. The overhead average for the last ten runs is -0.9% for Soot and 0.35 for JavaML.

The impact of MUSTI on the running time of real world Java program is low and less than 1%.

5.3 RQ3: Memory Overhead

To evaluate the memory overhead we analyze the classes shipped with Java 8 update 144 branch 1. These classes are the basic Java runtime classes such as `java.lang.String` which are present on any Java virtual machine and other classes such as the ones in packages `sun.*`, `com.sun.*` or `javax.*`. The total number of classes is 26,610. They represent approximately 160Mib.

We developed a program based on Soot to count the total number of instructions in all methods of all classes. For every constructor in a non-final non-private class, we add 3 instructions for every return instructions as well as 7 instructions representing the code we append to the constructors. For every non final non private method, we add 8 instructions representing the code we prepend to the methods.

In total there are 199,499 concrete methods representing 3,927,726 instructions. The overhead for constructors represents 213,395 instructions. The overhead for the methods represents 997,576 instructions. The total overhead for method instructions is 30.83%. If we assume that every instruction makes 10 bytes (an over-approximation), the memory necessary to represent the bytecode increases from 37 Mib to 49 Mib. The total size of all classes (including not only the bytecode of methods, but also the constant pools, the attributes, etc.) increases from 160 Mib to 172 Mib which represents an overhead of 7.5%.

The impact of MUSTI on the memory is low. New instructions add – at the maximum – an overhead of 7.5% which, in typical Java environments, represents only a few dozens of Mib.

5.4 RQ4: Vulnerable Code

Through this research question we aim at measuring the attack surface of the Java Class Library (JCL) for the invalid object initialization vulnerability. We search for vulnerable constructor methods and count them. To evaluate the number of vulnerable constructors, we developed a program based on Soot to statically analyze the constructors and extract the permissions they check. The analysis first constructs a CHA-based [17] call graph starting from the constructors' methods. Then, it searches the call graph down to a depth of 6 [8] for methods M checking for a permission (those are well defined in the Java documentation). For every M, it performs a context-sensitive backward analysis to extract the string representing the permission which is checked.

In total, we identified 938 constructors checking for 36 different kinds of permissions in 353 classes (1.33% of the 26,610 classes of the JDK). Notable vulnerable constructors are found in classes such as `java.lang.ClassLoader` (see Figure 1) where a permission is checked at a depth of 3, or in class `java.net.DatagramSocket` where a permission is checked at depth 4.

The number of classes actually checking for a permission in a constructor is small (1.33%) compared to the total number of classes in the JDK. This information can be used to optimize the number of classes and methods to instrument to reduce the runtime and memory overheads.

[8] we assume permissions are checked early in the constructor

6 Limitations

6.1 Bytecode Length Limit

The JVM restricts the bytecode size for a method to be less than 65,236 bytes [9]. In our experiments, we have never seen a method with a bytecode size greater than 25,744 bytes [9]. An attacker could craft such a method to prevent our approach from updating the bytecode. However, such a huge size for a method can be trivially detected and trigger a red flag to stop a potential attack. Furthermore, while our current implementation updates the bytecode, it could be updated to append native code when the bytecode is interpreted. This would make this problem of the bytecode size insignificant.

6.2 Field and Method Number Limit

The JVM restricts the number of methods to 65,535 and the number of fields to 65,535 [6]. Again, we have never seen classes with more than 1,260 methods [10] and 360 fields [11]. An attacker could craft such a class to prevent our approach from adding new fields. Nevertheless, this can be detected at runtime.

6.3 Other Attack Vectors

Our approach was designed to prevent invalid object initialization vulnerabilities. Other attacks based on buffer overflows, type confusion, confused deputy or other vulnerabilities which could also compromize the Java virtual machine sandbox are out of scope of this paper.

7 Discussion

In this section we first discuss two other approaches to prevent the vulnerability in Section 7.1. We then demonstrate that bypassing our mitigation technique would require a vulnerability more powerful than a invalid object initialization vulnerability in Section 7.2.

7.1 Other Approaches

There are other approaches than the one described in this paper to prevent the exploitation of invalid object initialization vulnerabilities. We discuss them here and highlight their advantages and drawbacks.

7.1.1 Hard-code Checks in Source Code

One approach which has already been partially implemented in the JCL is to explicitly hard-code checks in Java classes. While it prevents the vulnerability for beeing exploited in the updated class, the approach does not guarantee that all potentially vulnerable classes are protected. Furthermore, this approach adds *noise* to the code which makes it harder to read the code and to maintain it.

7.1.2 Patch the bytecode offline

The bytecode could be patched offline. That is all *.class* files could be modified to add code that will check for broken constructor chains. While this may reduce the overhead of loading classes, it also comes with limitations. First, only the bytecode of known classes can be modified and not the bytecode of classes created and loaded at runtime and loaded from the network. Second, the distribution of such code may break other programs which were not patched to support vulnerability check. Implementing the check directly in the JVM makes sure that the virtual machine state is consistent, i.e. that all classes are patched.

7.2 On the Possibility of Bypassing MUSTI

In this section, we demonstrate that bypassing MUSTI would require the use of a vulnerability that can bypass the Java sandbox. That is to say, our approach works unless there is a critical vulnerability which would allow the attacker to bypass all security checks of the sandbox including MUSTI.

7.2.1 Private Field Bypass

An analyst could try to set the field *is_initialized* of an *Object* instance to *true* even if the object has not been correctly initialized. If the analyst can do that, he has a primitive to break the encapsulation of private fields of Java objects. He can thus modify the `private static volatile SecurityManager security` private field of the `System` class to disable all permission checks. He can thus bypass all restrictions of the Java sandbox, which is absurd.

7.2.2 Removing Permission Checks

An analyst could try to remove permission checks from system classes. If the analyst can do that it means he has a primitive to modify the bytecode fo system classes which is a primitive more powerful than an exploit for an invalid object initialization. Thus, the analyst could

[9] `sun.awt.X11.XKeysym: void <clinit>()`

[10] `com.sun.corba.se.impl.logging.ORBUtilSystemException`

[11] `com.sun.tools.classfile.Opcode`

define code in system classes to bypass all restrictions of the Java sandbox including MUSTI, which is absurd.

8 Related Work

To the best of our knowledge, this work is the first presenting an approach to prevent invalid object initialization vulnerabilities.

Oh et al. [23] analyze CVE-2012-0507, a type confusion vulnerability, and explain how it has been used by malware.

Auriemma et al. [18] present techniques to bypass detection of known Java exploits by security tools. Techniques include serialization, splitting the exploit in multiple parts or leveraging multiple JVM.

Holzinger et al. [21] have studied public Java exploits. Their findings highlight that exploits leverage, among others, the following weaknesses of the Java platform: unauthorized use of restricted classes, arbitrary class loading and caller sensitivity. The paper presents the most vulnerable parts of Java but does not give any solution for preventing attacks based on the vulnerabilities.

Wang et al. [25] discuss simple techniques to detect Java exploits. One technique consists in disabling the security manager and run the suspicious Java code while checking if it tries to disable the security manager. If it does, there is a very high probability of it being a Java exploit. While most proof-of-concepts aim at disabling the security manager, not all practical attack actully need a security manager set to `null`.

Coker et al. [15] evaluates how the security manager is used in benign applications. Based on this knowledge, they devise two rules to prevent most of the exploits from working: the security manager cannot be changed if it has been set by the application and a class may not directly load a more privileged class if a security manager is set. This does not prevent malicious code from bypassing permission checks in constructors.

Holzinger et al. [20] presented an approach to remove shorcuts in stack-based access controls. While this improves the overall security of the JCL by making it much harder to have confused deputy attacks, it does not prevent attacks based on the vulnerability presented in this paper.

9 Conclusion

In this paper we have presented an approach, MUSTI, to prevent the exploitation of invalid object initialization vulnerabilities. From a security point of view it is essential to protect against this kind of critical vulnerability since it may allow to completely bypass the Java sandbox. We have evaluated our approach against an exploit leveraging a generic invalid object initialization vulnerability: it successfully prevents the exploits from working. The runtime and memory overhead have been evaluated using a Java benchmark and real world Java software. Our approach has a runtime overhead less than 5% and a memory overhead less than a dozen Mib which makes it practical.

Acknowledgements

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the VulFix project.

A Java Versions

For the sake of reproductibility, we list below the version of the programs/libraries/applications/files we used for our implementation and for the experiments.

- openjdk-8_8u144-b01-1.debian.tar.xz
  f0f94bd01397abdd966e64918bf3b350fc8c08b020eeeaf386d2dc76ff8554a7 (sha256)

- openjdk-8_8u144-b01.orig.tar.gz
  e816e1a8e2fee6ce21335cd8159805bde8e04be1c58214037cf39950fba991e5 (sha256)

- Soot commit
  cdef52ed39e849565e60609328017fe4885bd3d7

- Java-ML version 0.1.7

- DaCapo version 9.12

- Android application
  a02fe87870ece6e4772db1445670cfc5f06cf7cd5f646c457dac4eccb787e6be (sha256)

B Reverse of CVE-2017-3289

B.1 CVE-2017-3289

The description of the CVE indicates that *”Successful attacks of this vulnerability can result in takeover of Java SE, Java SE Embedded.”* [5]. This means it might be possible to exploit the vulnerability to escape the Java sandbox.

Redhat’s bugzilla indicates that *”An insecure class construction flaw, related to the incorrect handling of exception stack frames, was found in the Hotspot component of OpenJDK. An untrusted Java application or* *applet could use this flaw to bypass Java sandbox restrictions."* [1]. This informs the analyst that (1) the vulnerability lies in C/C++ code (Hotspot is the name of the Java VM) and that (2) the vulnerability is related to an illegal class construction and to exception stack frames. Information (2) indicates that the vulnerability is probably in the C/C++ code checking the validity of the bytecode. The page also links at the OpenJDK's patch for this vulnerability.

The OpenJDK's patch "8167104: Additional class construction refinements" fixing the vulnerability is available online [3]. Five C++ files are patched: `classfile/verifier.cpp`, the class responsible for verifying the structure and the validity of a class file, `classfile/stackMapTable.{cpp, hpp}`, the files handling the stack map table, and `classfile/stackMapFrame.{cpp, hpp}`, the files representing the stack map frames. In the following paragraph we first briefly describe what stack map frames and stack map table are and what is their use in bytecode verification. Next, we look at the patch to understand the vulnerability and present a proof of concept.

B.1.1 Bytecode Verification

Before loading a class, the JVM checks that all methods are valid. For instance, it checks that jumps land on valid instructions and not in the middle of an instruction and checks that the control flow ends with a return instruction. Furthermore, it also checks that instructions operate on valid types. Not correctly checking validity of types could lead to type confusion vulnerabilities. In Java, such vulnerabilities allow to bypass the sandbox. Type checking is thus a critical verification step.

Historically, to check type validity, the JVM relied on a data flow analysis to compute a fix point. This analysis may require to perform multiple pass over the same paths. As this is time consuming, and may slow down the class loading process, a new approach has been developed to perform the type checking in linear time where each path is only checked once. To achieve that, meta-information called *stack map frame*s have been added along the bytecode. In brief, stack map frames describe the possible types at each branch targets. Stack map frames are stored in a structure called the *stack map table* [8].

B.1.2 Looking at the Patch

By looking at the diff, one notices that function `StackMapFrame::has_flag_match_exception` has been removed and a condition, which we will refer to as $C1$, has been updated by removing the call to `has_flag_match_exception`. Also, methods `match_stackmap` and `is_assignable_to` have now one less parameter: `bool handler` has been removed. This parameter `handler` is set to `true` if the verifier is currently checking an exception handler.

Condition $C1$ is illustrated in Figure 15. This condition is within function `is_assignable_to` which checks if the current stack map frame is assignable to the target stack map frame, passed as a parameter to the function. Before the patch, the condition to return `true` was "match_flags `||` is_exception_handler `&&` has_flag_match_exception(target)". In English, this means that flags for the current stack map frame and the target stack map frame are the same or that the current instruction is in an exception handler and that function `has_flag_match_exception` returns `true`. Note that there exist only one kind of flag called `FLAG_THIS_UNINIT`. If this `UNINIT` flag is true, it indicates that the object referenced by "this" is uninitialized, i.e., its constructor has not yet been called.

After the patch, the condition becomes "match_flags". This means that, in the vulnerable version, there is probably a way to construct bytecode for which "match_flags" is false (i.e., "this" has the uninitialized flag in the current frame but not in the target frame), but for which `is_exception_handler` is `true` (the current instruction is in an exception handler) and for which `has_flag_match_exception(target)` returns `true`. But when does this function return `true`?

Function `has_flag_match_exception` is represented in Figure 16. In order for this function to return `true` all the following conditions must pass: (1) the maximum number of local variables and the maximum size of the stack must be the same for the current frame and the target frame (lines 4-5); (2) the current frame must have the `UNINIT` flag set to `true` (line 10); and (3) uninitialized objects are not used in the target frame (lines 14-24).

Figure 17 illustrates bytecode that satisfies the three conditions. (1) The maximum number of locals and the maximum stack size can be set to 2. (2) The current frame has `UNINIT` set to true (at line 7). (3) Uninitialized locals are not used in the target of the `athrow` instruction (line 11) since the first elements of the local is initialized to `TOP`. Note that the code has be within a try/catch block to have `is_exception_handler` set to `true` in function `is_assignable_to`. Moreover, notice that the bytecode is within a constructor (identified by the method name `<init>` in the bytecode). This is mandatory in order to have `FLAG_THIS_UNINIT` set to true.

Uninitialized "this". So What? At this point, the analyst is able to craft bytecode to return an uninitialized object in a constructor he controls. At first sight, it seems that the vulnerability does not bring anything to the analyst. However, looking closer we notice that this spe-

```
-   bool match_flags = (_flags | target->flags()) ==
      target->flags();
-   if (match_flags || is_exception_handler &&
      has_flag_match_exception(target)) {
+   if ((_flags | target->flags()) == target->flags())
      {
      return true;
    }
```

Figure 15: Patch for the vulnerability in the bytecode verifier.

```
bool StackMapFrame::has_flag_match_exception(
    const StackMapFrame* target) const {

  assert(max_locals() == target->max_locals() &&
         stack_size() == target->stack_size(), "
             StackMap sizes must match");

  VerificationType top = VerificationType::top_type()
      ;
  VerificationType this_type = verifier()->
      current_type();

  if (!flag_this_uninit() || target->flags() != 0) {
    return false;
  }

  for (int i = 0; i < target->locals_size(); ++i) {
    if (locals()[i] == this_type && target->locals()[
        i] != top) {
      return false;
    }
  }

  for (int i = 0; i < target->stack_size(); ++i) {
    if (stack()[i] == this_type && target->stack()[i]
        != top) {
      return false;
    }
  }

  return true;
}
```

Figure 16: Function has_flag_match_exception.

cially crafted bytecode could also be used in an analyst controlled constructor of a subclass to prevent the call to `super.<init>()`, the constructor of the super class. Since the super class could potentially be a class from the JCL, the vulnerability opens new perspectives to the analyst. The vulnerability allows the analyst to instantiate `public` classes for which the constructor checks a permission or for which the constructor is private.

B.2 Affected Versions

To our surprise, this vulnerability affects more than 40 different public releases. All Java 7 releases from update 0 to update 80 are affected. All Java 8 releases from update 5 to update 112 are also affected. Java 6 is not affected.

```
<init>()
  try_start:
    new "java/lang/Throwable"
    dup
    invokespecial "Throwable.<init>()"
    athrow
    // locals[0] = UNINITIALIZED_THIS
    // stack[0] = "java/lang/Throwable"
  try_end:
  handler:
    // locals[0] = TOP
    // stack[0] = "java/lang/Throwable"
    return
```

Figure 17: Proof-of-Concept for exploiting the uninitialized instance vulnerability in the bytecode verifier. In this example the constructor returns an instance that is still flagged with `UNINITIALIZED_THIS`.

B.3 Origin of the Vulnerability

By looking at the difference between the source code of the bytecode verifier of Java 6 update 43 and Java 7 update 0, we notice that the main part of the diff corresponds to the inverse of the patch presented in Figure 15. This means that the condition under which a stack frame is assignable to a target stack frame within an exception handler in a constructor has been weakened. Comments in the diff indicate that this new code has been added via request 7020118 [4]. This request asked to update the code of the bytecode verifier in such a way that NetBeans' profiler can generate handlers to cover the entire code of a constructor.

References

[1] Bug 1413562 - (cve-2017-3289) cve-2017-3289 openjdk: insecure class construction (hotspot, 8167104). Redhat `https://bugzilla.redhat.com/show_bug.cgi?id=1413562`.

[2] Lesson: Java applets. Oracle, `https://docs.oracle.com/javase/tutorial/deployment/applet/`.

[3] Openjdk changeset 8202:02a3d0dcbedd jdk8u121-b08 8167104: Additional class construction refinements. OpenJDK `http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/rev/02a3d0dcbedd`.

[4] Request for review (s): 7020118. `http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2011-February/001866.html`.

[5] Vulnerability summary for cve-2017-3289. National Vulnerability Database https://nvd.nist.gov/vuln/detail/CVE-2017-3289.

[6] The java virtual machine specification, java se 7 edition: 4.1. the classfile structure. Oracle https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1, 2013.

[7] The java virtual machine specification, java se 7 edition: 4.10.1. verification by type checking. Oracle https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.10.1, 2013.

[8] The java virtual machine specification, java se 7 edition: 4.7.4. the stackmaptable attribute . Oracle http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.4, 2013.

[9] The java virtual machine specification, java se 7 edition: 4.9. constraints on java virtual machine code. Oracle https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.9, 2013.

[10] The final countdown for npapi. Google, https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html, 2014.

[11] Npapi plugins in firefox. Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/, 2015.

[12] Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ml: A machine learning library. *Journal of Machine Learning Research*, 10(Apr):931–934, 2009.

[13] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[14] Cristina Cifuentes, Nathan Keynes, John Gough, Diane Corney, Lin Gao, Manuel Valdiviezo, and Andrew Gross. Translating java into llvm ir to detect security vulnerabilities. In *LLVM Developer Meeting*, 2014.

[15] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the flexibility of the java sandbox. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 1–10. ACM, 2015.

[16] Drew Dean, Edward W Felten, and Dan S Wallach. Java security: From hotjava to netscape and beyond. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 190–200. IEEE, 1996.

[17] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[18] Donato Ferrante and Luigi Auriemma. Reloading java exploits. In *Hack in the Box*, 2014.

[19] LSD Research Group et al. Java and java virtual machine security, vulnerabilities and their exploitation techniques. In *Black Hat Briefings*, 2002.

[20] Philipp Holzinger, Ben Hermann, Johannes Lerch, Eric Bodden, and Mira Mezini. Hardening java's access control by abolishing implicit privilege elevation. In *2017 IEEE Symposium on Security and Privacy (Oakland S&P)*, 2017.

[21] Philipp Holzinger, Stephan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS'16)*, 2016.

[22] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[23] Jeong Wook Oh. Recent java exploitation trends and malware. In *Black Hat USA*, 2012.

[24] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[25] Xinran Wang. An automatic analysis and detection tool for java exploits. In *Virus Bulletin*, 2013.