Errata for Security Vulnerability Notice

SE-2014-02-ORACLE-ERRATA

[Google App Engine Java security sandbox bypasses, Issue 42]

2011 Security Explorations. All Rights Reserved.

DISCLAIMER

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NEITHER SECURITY EXPLORATIONS, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE INFORMATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THERE IS NO WARRANTY BY SECURITY EXPLORATIONS OR BY ANY OTHER PARTY THAT THE INFORMATION CONTAINED IN THE THIS DOCUMENT WILL MEET YOUR REQUIREMENTS OR THAT IT WILL BE ERROR-FREE.

YOU ASSUME ALL RESPONSIBILITY AND RISK FOR THE SELECTION AND USE OF THE INFORMATION TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE INSTALLATION, USE, AND RESULTS OBTAINED FROM IT.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SECURITY EXPLORATIONS, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION CONTAINED IN THIS DOCUMENT, EVEN IF SECURITY EXPLORATIONS OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS.

2011 Security Explorations. All Rights Reserved.

INTRODUCTION

On Jun 30, 2015 Security Explorations reported a security vulnerability (Issue 42) to Oracle affecting Java SE 7 [1].

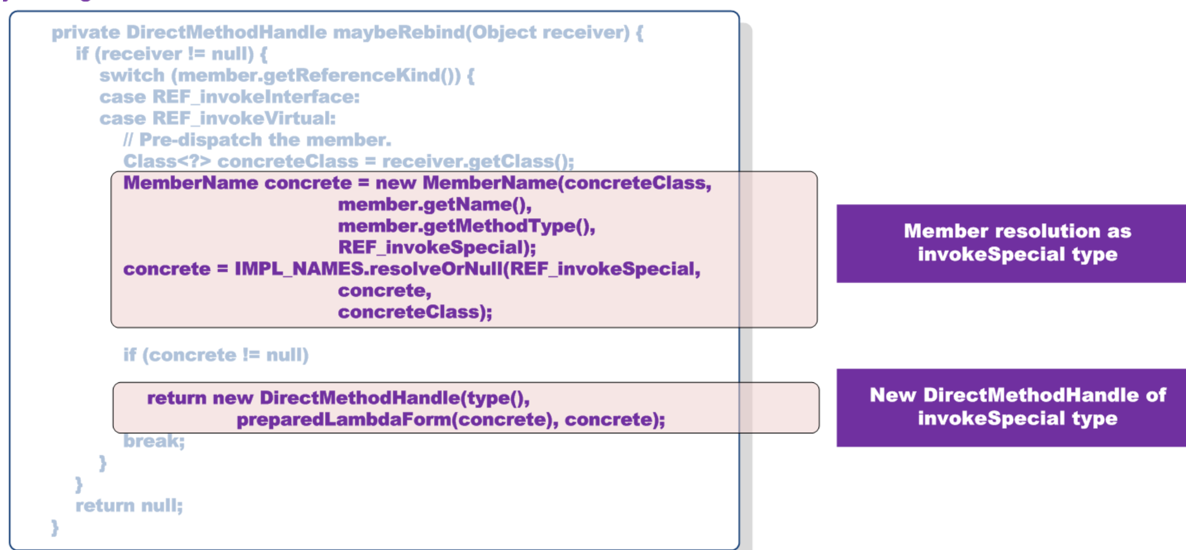
In our original report [2], we indicated that the vulnerability had its origin in `klassItable::initialize_itable_for_interface` method's implementation of Java SE 7 HotSpot VM. We have recently learned that our initial analysis regarding the root cause of Issue 42 was incorrect.

Below, we provide more detailed information about the actual cause of the vulnerability, the reasoning that has mislead us into concluding that Issue 42 was caused by an improper initialization of non-public interface method slots and some additional findings regarding this issue.

THE REAL ROOT CAUSE

The actual cause of Issue 42 (assigned CVE-2015-4871) lies in the possibility to convert a `DirectMethodHandle` denoting an interface method to a method handle indicating a special instance method (the method invoked with the use of an *invokespecial* instruction). This can in particular occur at the time of a binding[1] the receiver of a target method handle. As a result, protected instance methods could be successfully used (and called) as interface methods. The reason is the *invokespecial* bytecode instruction which can access such methods (`invokeinterface` cannot).

The abovementioned method handle conversion takes place in a private `maybeRebind` method of `java.lang.invoke.DirectMethodHandle` as illustrated on Fig. 1:



DirectMethodHandle conversion changes method dispatch from *invokeinterface* to *invokespecial* based one

Fig. 1 Method handle conversion exploited by Issue 42.

The `maybeRebind` method can be reached as part of a `bindTo` method invocation chain, which is illustrated on Fig. 2.

[1] with the use of a `bindTo` method of `java.lang.invoke.MethodHandle` class.

2011 Security Explorations. All Rights Reserved.

java.lang.invoke.MethodHandle

```
public MethodHandle bindTo(Object x) {
    Class<?> ptype;
    MethodType type = type();
    if (type.parameterCount() == 0 || (ptype = type.parameterType(0)).isPrimitive())
              throw newIllegalArgumentException(
                                          "no leading reference parameter", x);
    x = ptype.cast(x);
    return bindReceiver(x);
}
```

java.lang.invoke.DirectMethodHandle

```
MethodHandle bindReceiver(Object receiver) {
    DirectMethodHandle concrete = maybeRebind(receiver);
    if (concrete != null)
        return concrete.bindReceiver(receiver);
    return super.bindReceiver(receiver);
}
```

INVOCATION OF VULNERABLE METHOD

Fig. 2 The `bindTo` invocation chain leading to `maybeRebind` method.

The root cause of Issue 42 can be easily confirmed by running our POC code [3] with `DEBUG_NAMES` method handles API debugging property enabled[2]. It provides the following output if run under vulnerable Java SE 7 software:

```
MethodHandle(MyINTF)void/LF=DMH.invokeInterface_L_V=Lambda(a0:L,a1:L)=>{
    t2:L=DirectMethodHandle.internalMemberName(a0:L);
    t3:V=MethodHandle.linkToInterface(a1:L,t2:L);t3:V}/
                    DMH=Test$MyINTF.setError()void/invokeInterface
Test$MyPrintStream@986b0ee
checkError(): false
MethodHandle()void/LF=BMH.reinvoke=Lambda(a0:L)=>{
    t1:L=BoundMethodHandle$Species_LL.argL1(a0:L);
    t2:L=MethodHandle.reinvokerTarget(a0:L);
    t3:V=MethodHandle.invokeBasic(t2:L,t1:L);void}/
                    BMH=[MethodHandle(MyINTF)void
/LF=DMH.invokeSpecial_L_V=Lambda(a0:L,a1:L)=>{
    t2:L=DirectMethodHandle.internalMemberName(a0:L);
    t3:V=MethodHandle.linkToSpecial(a1:L,t2:L);t3:V}/
                    DMH=java.io.PrintStream.setError()void/invokeSpecial,
                                          Test$MyPrintStream@986b0ee]
checkError(): true
```

The above clearly shows that a type of a target `DirectMethodHandle` (DMH) gets changed from `invokeInterface` to `invokeSpecial`. At the same time the interface method handle is changed, so that it denotes a target instance method.

THE FAILED REASONING

Our original Proof of Concept Code (POC) was developed in Java SE 7 environment as this Java version was in use by Google App Engine at the time of our investigation (May / Jun 2015).

We verified that the POC didn't work under Java SE 8. In order to locate the root cause of the vulnerability, we proceeded with a more detailed investigation of the reasons for the failure of the POC in Java SE 8.

---

[2] with `-Djava.lang.invoke.MethodHandle.DEBUG_NAMES=true` argument passed to JVM.

2011 Security Explorations. All Rights Reserved.

We noticed that in Java SE 8, the invocation of a protected instance method through an interface method handle triggered `IllegalAccessError`. This error was raised by `throwIllegalAccessError` method of `sun.misc.Unsafe` class:

```
java.lang.IllegalAccessError
      at sun.misc.Unsafe.throwIllegalAccessError(Unsafe.java:1139)
      at Test$MyPrintStream.invoke_interface(Test.java:58)
      at Test.main(Test.java:69)
```

We inspected OpenJDK 8 source code and discovered that `throwIllegalAccessError` method of `sun.misc.Unsafe` class was invoked only from one code location. This was the `klassItable::initialize_itable_for_interface` method and its part handling initialization of interface method tables entries corresponding to invalid (such as non-public) interface methods in particular:

```
if (target == NULL || !target->is_public() || target->is_abstract()) {
// Entry does not resolve. Leave it empty for AbstractMethodError.
      if (!(target == NULL) && !target->is_public()) {
      // Stuff an IllegalAccessError throwing method in there instead.
            itableOffsetEntry::method_entry(
                  _klass(), method_table_offset)[m->itable_index()].
                  initialize(Universe::throw_illegal_access_error());

      }
}
```

At the same time, we discovered that a code sequence corresponding to Java SE 8 location where a slot of a non-public interface method was filled with a pointer to the method throwing an *IllegalAccessError* was missing in OpenJDK 7 code:

```
methodOop target = klass->uncached_lookup_method(
                                   method_name, method_signature);
...
if (target == NULL || !target->is_public() || target->is_abstract()) {
  // Entry do not resolve. Leave it empty
}
```

In the next step, we decided to verify whether Issue 42 had its origin in `klassItable::initialize_itable_for_interface` method's implementation. We considered the following two options to proceed with:

1) recompilation of OpenJDK 7 software with an additional code sequence corresponding to Java SE 8 code (filling the `IllegalAccessError` throwing method in the invalid interface table entry),
2) modification of Java SE 8 binary in a way that would "mimic" Java SE 7 code behavior (no `IllegalAccessError` throwing method in the invalid interface table entry).

We chose the second option as it was more easier / faster to proceed with. We kept in mind that a backport of method handles implementation was made from Java SE 8 to Java SE 7[3]. As a result, we assumed that the implementation of method handles API would not differ much between Java SE 7 and 8 versions. At that time, we were not aware of inconsistent

[3] on 26-Jul-2013 Oracle informed us that Issue 69 of SE-2012-01 project would be addressed by a backported (from JDK 8) implementation of the affected component (method handles API) in JDK 7 Update 40 [4].

changes being applied to their code bases (that their implementation could differ at core Java classes level in particular). This reasoning and approach taken is illustrated on Fig. 3.

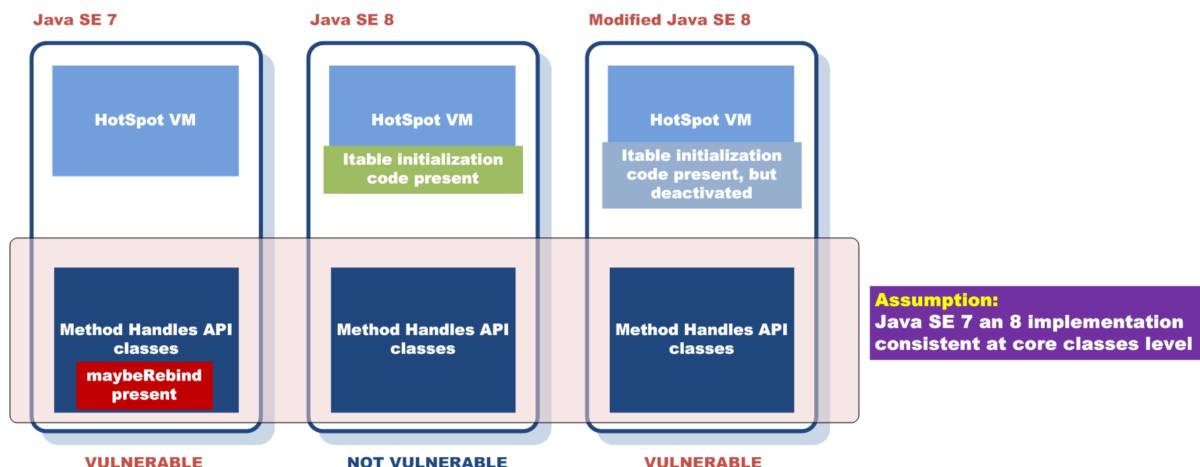


Fig. 3 The illustration of a wrong assumption leading to the incorrect root cause analysis.

We modified `jvm.dll` binary of Java SE 8 Update 45 software, so that a code sequence implementing the initialization of an interface table method entry with an `IllegalAccessError` throwing method was always skipped. We accomplished that by applying the following change to Java VM code:

```
original code sequence:
loc_704E555A:
            mov     eax, [rdx+20h]
            movzx   ecx, al
            and     cl, 1            ; JVM_ACC_PUBLIC   0x0001
            jz      loc_704E568F     ; -> !target->is_public()
            shr     eax, 0Ah         ; JVM_ACC_ABSTRACT 0x0400
            test    al, 1
            jnz     loc_704E568F     ; -> target->is_abstract()
                  ...
loc_704E568F:
            test    cl, cl
            jnz     short loc_704E56C1
            mov     rdi, cs:Universe___throw_illegal_access_error
            test    rdi, rdi
            jz      short loc_704E56C1
            movsxd  rax, dword ptr [rsi+24h]
            movsxd  rcx, dword ptr [rbp+67h]
            lea     rax, ds:50h[rax*8]
            sub     rcx, rax
            mov     rax, [rbp+5Fh]
            mov     rax, [rax]
            mov     [rcx+rax], rdi

patched code sequence:
loc_704E555A:
            mov     eax, [rdx+20h]
            movzx   ecx, al
            and     cl, ffh          ; PATCHED INSTRUCTION
            jz      loc_704E568F     ; -> !target->is_public()
```

```
                shr     eax, 0Ah         ; JVM_ACC_ABSTRACT 0x0400
                test    al, 1
                jnz     loc_704E568F     ; -> target->is_abstract()
                            ...
loc_704E568F:
```

We discovered that the patched JVM successfully processed the invocation of protected methods through an interface method handle (no `IllegalAccessError` was thrown).

All of the above has lead us to the wrong conclusion that Issue 42 was caused by an improper initialization of a non-public interface method slots.

ADDITIONAL FINDING

We decided to investigate this further in order to see when and why things started to differ between Java SE 7 and 8 versions with respect to the flawed implementation of `DirectMethodHandle` class.

We found out that the vulnerable code (`bindArgument`, `bindReceiver` and `maybeRebind` methods of `java.lang.invoke.DirectMethodHandle` class) was present in Oracle Java SE 8 till version 8 Update 31. It was not available any more in Java SE 8 Update 40 released on Mar 3, 2015 [5]. The corresponding change was not however applied to Oracle Java SE 7 code regardless of the mirror implementation of `java.lang.invoke.DirectMethodHandle` class[4].

We also inspected the OpenJDK source code and found out that a vulnerable code was removed from OpenJDK 8 on Sep 10, 2014. This was done under the changeset associated with bug id 8050166 [6] and annotated as "*Get rid of some package-private methods on arguments in j.l.i.MethodHandle*" [7].

At the time of investigating a fix applied to OpenJDK 7 (OpenJDK bug id 8142882 [8]) and addressing Issue 42 [9], we discovered that it was a mirror of the abovementioned changeset from Sep 10, 2014. This indicates that CVE-2015-4871 would not exist if a changeset from OpenJDK 8 was backported to OpenJDK 7 (if consistent changes were made across Java SE 7 and 8 code bases).

FINAL WORDS

The case of incorrect analysis regarding the root cause of Issue 42 has reminded us that in the world of security vulnerabilities, incorrect or unverified assumptions can easily lead to wrong conclusions. These are usually the software vendors that fall victim of it. As it turns out, the very same can happen to security researchers.

Java SE 7 and 8 runtimes may look similar, but the devil lies in the details. Some code changes applied to their codebases are introduced in an inconsistent manner. As a result, the underneath implementation of key Java VM features including core classes can be different across Java SE 7 and 8. This can sometimes influence security of software and

[4] it also contained `bindArgument`, `bindReceiver` and `maybeRebind` methods.

2011 Security Explorations. All Rights Reserved.

either introduce new or leave existing vulnerabilities in code. This can also make their root cause analysis very tricky of which CVE-2015-4871 is a perfect example.

REFERENCES

[1] SE-2014-02 Vendors status
`http://www.security-explorations.com/en/SE-2014-02-status.html`
[2] SE-2014-02-ORACLE, Issue #42
`http://www.security-explorations.com/materials/SE-2014-02-ORACLE.pdf`
[3] Proof of Concept code for Issue 42
`http://www.security-explorations.com/materials/se-2014-02-42.zip`
[4] SE-2012-01 Vendors status
`http://www.security-explorations.com/en/SE-2012-01-status.html`
[5] Java version history
`https://en.wikipedia.org/wiki/Java_version_history`
[6] JDK-8050166
`https://bugs.openjdk.java.net/browse/JDK-8050166`
[7] OpenJDK changeset for JDK-8050166 (src/share/classes/java/lang/invoke/DirectMethodHandle.java)
`http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/diff/c7be76a1dda5/src/share/classes/java/lang/invoke/DirectMethodHandle.java`
[8] JDK-8142882
`https://bugs.openjdk.java.net/browse/JDK-8142882`
[9] OpenJDK changeset for JDK-8142882 (src/share/classes/java/lang/invoke/DirectMethodHandle.java)
`http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/diff/c434c67b8189/src/share/classes/java/lang/invoke/DirectMethodHandle.java`

About Security Explorations

Security Explorations (`http://www.security-explorations.com`) is a security start-up company from Poland, providing various services in the area of security and vulnerability research. The company came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 50 security issues uncovered in the Java technology over the recent years. He is also the hacking contest co-winner and the man who has put Microsoft Windows to its knees (vide MS03-026). He was also the first one to present successful and widespread attack against mobile Java platform in 2004.

2011 Security Explorations. All Rights Reserved.