Android Serialization Vulnerabilities Revisited

Roee Hay
X-Force Application Security Team Lead
IBM Security
@roeehay

Joint work with Or Peles
We will see how this Android SDK class

```java
public class OpenSSLX509Certificate extends X509Certificate {
    private final long mContext;
    ...
}
```

MISSING MODIFIER
BEFORE OUR DISCLOSURE!
(NOW PATCHED)
Led to malware capable of this...

- REPLACEMENT OF APPS
- SELinux BYPASS
- ACCESS TO APPS’ DATA
- KERNEL CODE EXEC (on select devices)
Introduction
Serialization

class foo {
    int bar; = 1234
    String baz; = “hi”
    long *qux; = 0xf334233
}
Serialization

**SENDER**

class foo {
    int bar; = 1234
    String baz; = “hi”
    long *qux; = 0xf334233
}

**MEDIA**

001010...0110

**RECIPIENT**

Serialize
Serialization

**SENDER**

class foo {
    int bar; = 1234
    String baz; = “hi”
    long *qux; = 0x1f334233
}

**RECIPIENT**

class foo {
    int bar; = 1234
    String baz; = “hi”
    long *qux; = 0x14d3e2c3
}

**MEDIA**

001010...0110

**Serialize** → **MEDIA** → **Deserialize**
Vulnerability Root Cause

DESERIALIZATION OF UNTRUSTED DATA

ObjectInputStream ois =
    new ObjectInputStream(insecureSource);

Foo t = (Foo)ois.readObject();
ATTACKER

```java
class dangerous {
    ...
}
```

VICTIM

```java
class dangerous {
    ...
}
```

**MEDIA**

```
001010...0110
```

Serialized data transmitted to **VICTIM**.
Example of a vulnerability

ATTACKER

```cpp
class dangerous {
    int *ptr;
}
```

VICTIM

```cpp
class dangerous {
    int *ptr;
}
```

MEDIA

```
001010...0110
```

Serialize

Deserialize
Example of a vulnerability

ATTACKER

class dangerous
{
    int *ptr; = 0x66666666
}

VICTIM

class dangerous
{
    int *ptr; = 0x66666666
}

MEDIA

Serialize

001010...0110

Deserialize
ObjectInputStream ois =
    new ObjectInputStream(insecureSource);

dangerous t = (dangerous)ois.readObject();

callNative(t.ptr)
History of Serialization Vulnerabilities

2009 - Shocking News in PHP Exploitation
2011 - Spring Framework Serialization-based remoting vulnerabilities
2012 - AtomicReferenceArray type confusion vulnerability
2013 - Apache Commons FileUpload Deserialization Vulnerability
        - Ruby on Rails YAML Deserialization Code Execution
2014 - Android <5.0 Privilege Escalation using ObjectInputStream
2015 - Android OpenSSLX509Certificate Deserialization Vulnerability
        - Apache Groovy Deserialization of Untrusted Data
        - Apache Commons Collections Unsafe Classes
<table>
<thead>
<tr>
<th>Year</th>
<th>Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>- Shocking News in PHP Exploitation</td>
</tr>
<tr>
<td>2011</td>
<td>- Spring Framework Serialization-based remoting vulnerabilities</td>
</tr>
<tr>
<td>2012</td>
<td>- AtomicReferenceArray type confusion vulnerability</td>
</tr>
<tr>
<td>2013</td>
<td>- Apache Commons FileUpload Deserialization Vulnerability</td>
</tr>
<tr>
<td></td>
<td>- Ruby on Rails YAML Deserialization Code Execution</td>
</tr>
<tr>
<td>2014</td>
<td>- Android &lt;5.0 Privilege Escalation using ObjectInputStream</td>
</tr>
<tr>
<td>2015</td>
<td>- Android OpenSSLX509Certificate Deserialization Vulnerability</td>
</tr>
<tr>
<td></td>
<td>- Apache Groovy Deserialization of Untrusted Data</td>
</tr>
<tr>
<td></td>
<td>- Apache Commons Collections Unsafe Classes</td>
</tr>
</tbody>
</table>
Android Inter-App Communication
Android Inter-App Communication 101

Intent { play:/// ...}

Intent { sms:/// ...}
An Intent can also contain

- Bundle
  - Strings
  - Primitives
  - Arrays
An Intent can also contain

Bundle

- Strings
- Primitives
- Arrays
- Serializable Objects
Motivation
Previous work

CVE-2014-7911 (Jann Horn):

Non-Serializable Classes can be Deserialized on target.
Step 1. Find an interesting target.
The Target

SYSTEM_SERVER
Step 2. Send target a ‘serialized’ object in a Bundle

MALWARE

SYSTEM_SERVER

Serialized Object of non-Serializable class
final class BinderProxy implements IBinder {

    private long mOrgue;  // POINTER
    ... 
    private native final destroy();

    @Override
    protected void finalize() throws Throwable{
        try {
            destroy();
        } finally {
            super.finalize();
        } 
    }
}
Android Apps are not just Java...

App

Java

Native (C/C++)
Android Apps are not just Java…

Java

Native (C/C++)

JNI

App
Pointers may pass back and forth

App

JNI

Java

Native (C/C++)

pointers
Pointers may pass back and forth

App

Java

Native (C/C++)

JNI

pointers
final class BinderProxy implements IBinder {

    private long mOrgue; \rightarrow \text{POINTER}

    ... 

    private native final destroy();

    @Override
    protected void finalize() throws Throwable {
        try {
            destroy();
        }
        finally {
            super.finalize();
        }
    }
}
Step 3. Make it deserialize on the target

MALWARE → SYSTEM_SERVER

Deserialized object
All Bundle members are deserialized with a single ‘touch’ without **type checking** before deserialization

e.g.

```java
public String getString(String key) {
    parcel(); // DESerializes ALL
    final Object o = mMap.get(key);
    try { return (String) o; }
    catch (ClassCastException e) {
        typeWarning(...)
    }
}
```
Step 4. Make one of its methods *execute* on target.
final class **BinderProxy** implements IBinder {

    private long mOrgue;
    private native final destroy();

    @Override
    protected void finalize() throws Throwable {
        try { destroy(); }  
        finally { super.finalize(); }
    }

← EXECUTED AUTOMATICALLY BY THE GC
A Word about Garbage Collection

App’s Memory

Object A (root)

Object B

Object C
A Word about Garbage Collection

App’s Memory

Object A
(root)

Object B

Object C
A Word about Garbage Collection

App’s Memory

Object A
(root)

Object B
final class BinderProxy implements IBinder {

    private long mOrgue;
    ...

    private native final destroy();

    @Override
    protected void finalize() throws Throwable {
        try {
            destroy();
        } finally {
            super.finalize();
        }
    }

    \longrightarrow\text{EXECUTED AUTOMATICALLY BY THE GC}
final class **BinderProxy** implements IBinder {

private long mOrgue;
…
private native final destroy(); ➔ NATIVE METHOD THAT USES THE PTR

@Override
protected void finalize() throws Throwable
{
    try { destroy(); }
    finally { super.finalize(); }
}
}
Google’s Patch for CVE-2014-7911

Do not Deserialize Non-Serializable Classes
Our 1st contribution: The Android Vulnerability

CVE-2015-3825/37
Class Foo implements `Serializable` {

    private long mObject;
    ...
    private native final destroy();

    @Override
    protected void finalize() throws Throwable {
        try {
            destroy();
        }
        finally {
            super.finalize();
        }
    }
}
Experiment 1
Experiment 1

boot.art
Experiment 1

boot.art

~13K Loadable Java Classes
Experiment 1

boot.art

App: Loaded classes using Reflection

~13K Loadable Java Classes
Experiment 1

boot.art

~13K Loadable Java Classes

App: Loaded classes using Reflection

Dumped classes:
1. Serializable
2. Finalize method
3. Controllable fields
The Result

OpenSSLX509Certificate
public class OpenSSLX509Certificate extends X509Certificate {

    private final long mContext;

    @Override
    protected void finalize() throws Throwable {
        ...

        NativeCrypto.X509_free(mContext);
        ...
    }
}
public class OpenSSLX509Certificate extends X509Certificate {

    private final long mContext;

    @Override
    protected void finalize() throws Throwable {
        ...
        NativeCrypto.X509_free(mContext);
        ...
    }
}
public class OpenSSLX509Certificate extends X509Certificate {

    private final long mContext;

    @Override
    protected void finalize() throws Throwable {
        ...
        NativeCrypto.X509_free(mContext);
        ...
    }
}

(1) SERIALIZABLE
(2) CONTROLLABLE POINTER
public class OpenSSLX509Certificate extends X509Certificate {

    private final long mContext;

    @Override
    protected void finalize() throws Throwable {
        ... NativeCrypto.X509_free(mContext); ... 
    }
}
NativeCrypto.X509_free(mContext)

X509_free(x509);  // x509 = mContext

ASN1_item_free(x509, ...)

asn1_item_combine_free(&val, ...);  // val = *pval = mContext

if (asn1_do_lock(pval, -1,...) > 0)
    return;

// Decreases a reference counter (mContext+0x10)
// MUST be POSITIVE INTEGER (MSB=0)
ref = mContext+0x10
if (*ref > 0)
    *ref--
else
    free(…)

Arbitrary Decrement
Proof-of-Concept Exploit

Arbitrary Code Execution in system_server
Exploit Outline

MALWARE

Malicious Serialized Object(s) w/ payload buffer

SYSTEM_SERVER
Exploit Outline

MALWARE

SYSTEM_SERVER

shellcode
First Step of the Exploit

Own the Program Counter (PC)
Own the Program Counter

1. Override some offset / function ptr
2. Get it called.
Creating an Arbitrary Code Exec Exploit

ARSENAL

1. Arbitrary Decrement
2. Controlled Buffer
Constrained Arbitrary Memory Overwrite

Bundle

OpenSSLX509Certificate
mContext=0x11111100

0x11111110 − = 1
Constrained Arbitrary Memory Overwrite

Bundle

OpenSSLX509Certificate
mContext=0x11111100

OpenSSLX509Certificate
mContext=0x11111100

\[ * 0x11111110 - = 2 \]
Constrained Arbitrary Memory Overwrite

Bundle

OpenSSLX509Certificate
mContext=0x11111100

OpenSSLX509Certificate
mContext=0x11111100

* 0x111111110 −= n
Constrained Arbitrary Memory Overwrite

**Bundle**

- OpenSSLX509Certificate
  - mContext=0x11111100

- OpenSSLX509Certificate
  - mContext=0x11111100

- ...

- OpenSSLX509Certificate
  - mContext=0x11111100

* $0x11111110 \equiv n$

and If we knew the original value:

**Arbitrary Overwrite**
Creating an Arbitrary Code Exec Exploit

ARSENAL

1. Arbitrary Decrement
2. Controlled Buffer
3. Arbitrary Overwrite
   (if we knew the original value)
Creating an Arbitrary Code Exec Exploit

**ARSENAL**
1. Arbitrary Decrement
2. Controlled Buffer
3. Arbitrary Overwrite
   (if we knew the original value)

**DEFENSES**
1. ASLR
2. RELRO
3. NX pages
4. SELinux
Finding the original value: observation

**system_server**

```
root@generic:/# cat /proc/<system_server>/maps

70e40000-72cee000 r-p ... boot.oat
72cee000-74400000 r-xp ... boot.oat
74400000-74401000 rw-p ... boot.oat
...
aa09f000-aa0c3000 r-xp ... libjavacrypto.so
aa0c3000-aa0c4000 r--p ... libjavacrypto.so
aa0c4000-aa0c5000 rw-p ... libjavacrypto.so
...
bb650000-b66d5000 r-xp ... libcrypt.so
bb66d6000-b66e1000 r--p ... libcrypt.so
bb66e1000-b66e2000 rw-p ... libcrypt.so
```
fork without execve = no ASLR!
Determining the value

Zygote

system_server

fork()  

fork()  

fork()  

malware

<libXYZ> value  

<libXYZ> value  

<libXYZ> value
Creating an Arbitrary Code Exec Exploit

**ARSENAL**

1. Arbitrary Decrement
2. Controlled Buffer
3. Arbitrary Overwrite *(if we knew the original value)*

**DEFENSES**

1. ASLR
2. RELRO
3. NX pages
4. SELinux
Using the Arbitrary Overwrite

Goal.

Overwrite some pointer

Problem.

.got is read only (RELRO)
A function pointer under .data

`id_callback` in `libcrypto`

Called during deserialization of:

`OpenSSLECPrivateKey`
Triggering id_callback remotely

Malware

Bundle

OpenSSLECPriKey
BAD DATA
that leads to the right path

system_server
First Step Accomplished

We now own the Program Counter
Creating an Arbitrary Code Exec Exploit

ARSENAL

1. Arbitrary Decrement
2. Controlled Buffer
3. Arbitrary Overwrite
   (if we knew the original value)

DEFENSES

1. ASLR
2. RELRO
3. NX pages
4. SELinux
Next Steps of the PoC Exploit (simplified)

- pc → r-x code
- sp → rw- stack
- rw- ROP chain
- rw- shellcode

system_server
Problem 1: SP does not point at ROP chain

- pc → r-x code
- sp → rw- stack
- rw- ROP chain
- rw- shellcode

system_server
Solution: Stack Pivoting

Our buffer happens to be pointed by fp. The Gadget: \texttt{mov sp, fp; ...; pop {...}}

\begin{itemize}
  \item \texttt{pc \rightarrow r-x code/pivot}
  \item \texttt{sp \rightarrow rw- stack}
  \item \texttt{fp \rightarrow rw- ROP chain}
  \item \texttt{rw- shellcode}
\end{itemize}
Solution: Stack Pivoting

Our buffer happens to be pointed by fp.
The Gadget: `mov sp, fp; ..., pop {...}`

Gadget: Stack Pivot

system_server

pc → r-x code/pivot
rw- stack
sp → rw- ROP chain
rw- shellcode
Allocating RWX Memory

Gadget: Stack Pivot

Gadget: mmap/RWX

system_server

pc → r-x code/mmap
sp → rw- stack
fp → rw- ROP chain
rw- shellcode
Problem 2: SELinux should prohibit mmap/RWX

Gadget: Stack Pivot

Gadget: mmap/RWX

system_server

pc → r-x code/mmap
sp → rw- stack
fp → rw- ROP chain
rw- shellcode
Solution: Weak SELinux Policy for system_server

Gadget: Stack Pivot

Gadget: mmap/RWX

system_server

pc → r-x code/mmap
sp → rw- stack
fp → rw- ROP chain
rw- shellcode
Solution: Weak SELinux Policy for system_server

allow system_server self:process execmem
Allocating RWX Memory

Gadget: Stack Pivot

Gadget: mmap/RWX

system_server

pc → r-x code/mmap
rw- stack
sp → rw- ROP chain
rw- shellcode
rwx -
Copying our Shellcode

Gadget: Stack Pivot
Gadget: memcpy
Gadget: mmap/RWX

system_server
pc → r-x code/memcpy
rw- stack
sp → rw- ROP chain
rw- shellcode
rwx -
Copying our Shellcode

Gadget: Stack Pivot

Gadget: memcpy

Gadget: mmap/RWX

system_server

pc → r-x code/memcpy
rw- stack
sp → rw- ROP chain
rw- shellcode
rwx shellcode
Executing our Shellcode

system_server

r-x code
rw- stack
sp → rw- ROP chain
rw- shellcode
pc → rwx shellcode

Gadget:
Stack Pivot

Gadget:
mmap/RWX

Gadget:
memcpy

shellcode
Creating an Arbitrary Code Exec Exploit

**ARSENAL**

1. Arbitrary Decrement
2. Controlled Buffer
3. Arbitrary Overwrite
   (if we knew the original value)

**DEFENSES**

1. ASLR
2. RELRO
3. NX pages
4. SELinux
Shellcode

Runs as system, still subject to the SELinux, but can:

- REPLACEMENT OF APPS
- SELINUX BYPASS
- ACCESS TO APPS’ DATA
- KERNEL CODE EXEC
  (on select devices)
Demo
Google’s Patch for CVE-2015-3825

```java
public class OpenSSLX509Certificate extends X509Certificate {
    private transient final long mContext;
    ...
}
```

MISSING MODIFIER BEFORE OUR DISCLOSURE!
(NOW PATCHED)
Hardened SELinux policy in the AOSP master branch

AOSP Commit #1:

```
commit 23cde8776b94ff2228f3a8d845d4
author Nick Kralevich <nnk@google.com>
commiter Nick Kralevich <nnk@google.com>
tree e02b4bf1bd6d09223b7cfa7af13fa178c286ad5b
parent arfd1d3c045d6b79386a50be58972ae6bc11c

system_server: remove old dalvik JIT rules on user/userdebug builds

On user and userdebug builds, system_server only loads executable content from /data/dalvik-cache and /system. JITting for system_server is only supported on eng builds. Remove the rules for user and userdebug builds.

Going forward, the plan of record is that system_server will never use JIT functionality, instead using dex2oat or interpreted mode.

Inspired by https://android-review.googlesource.com/#/c/39944

Change-Id: I Addsacragx672005268beafed22167c6013582
```

AOSP Commit #2:

```
commit 82bdd796e1265bd0e4b0497e9be
author Nick Kralevich <nnk@google.com>
commiter Nick Kralevich <nnk@google.com>
tree 045a66c2b97adfb7b0b19b4e4eeb66e25215a6d
parent del1f5817c53aabb22124254856d2d1148fd2f6

system_server: (eng builds) remove JIT capabilities

23cde8776b94ff2228f3a8d845d41052af52319e removed JIT capabilities from system_server for user and userdebug builds. Remove the capability from eng builds to be consistent across build types.

Add a neverallow rule (compile time assertion + CTS test) to verify this doesn't regress on our devices or partner devices.

Bug: 2346888
Bug: 2491526
Change-Id: I Addsacragx672005268beafed22167c6013582
```
Good news!
Majority of the devices are updated

But some aren’t…
Our 2nd Contribution: Vulnerabilities in SDKs

Finding Similar Vulnerabilities in SDKs

Goal. Find vulnerable Serializable classes in 3\textsuperscript{rd}-party SDKs

Why. Fixing the Android Platform Vulnerability is not enough. Apps can be exploited as well!
Experiment 2

Analyzed over 32K of popular Android apps

Main Results

<table>
<thead>
<tr>
<th>CVE</th>
<th>SDK</th>
<th>Code Exec.</th>
</tr>
</thead>
</table>
SWIG, a C/C++ to Java interoperability tool, can generate vulnerable classes.
Patching is problematic for SDKs
Apps are in a bad place

- **Vulnerable apps are still out there.**
  - SDKs need to be updated by app developers.
  - Dozens of apps still use them! (as of Feb ‘16)

- **New vulnerable apps can emerge**
  - Developers can introduce their own vulnerable classes.
Apps are in a bad place

- **Exploitation**
  - Still no *type-checking* by Android before deserialization.
  - **ASLR** can still be defeated when malware attacks Zygote forked processes.
  - As opposed to system_server, The **SELinux** policy hasn’t been hardened for the **apps domain**.

```
9  # WebView and other application-specific JIT compilers
10  allow appdomain self:process execmem;
```
Wrap-up
Summary

- Found a high severity vulnerability in Android (Exp. 1).
- Wrote a reliable PoC exploit against it.
- Found similar vulnerabilities in 6 third-party SDKs (Exp. 2).
- Patches are available for all of the vulnerabilities and also for SWIG.
  - **Consumers:** Update your Android.
  - **Developers:**
    - Update your SDKs.
    - Do not create vuln. Serializable classes. Use *transient* when needed!
Are you still vulnerable?
References

- **Paper.** [https://www.usenix.org/conference/woot15/workshop-program/presentation/peles](https://www.usenix.org/conference/woot15/workshop-program/presentation/peles)

- **Video.** [https://www.youtube.com/watch?v=VekzwVdwqIY](https://www.youtube.com/watch?v=VekzwVdwqIY)


- **AOSP Patch.** [https://android.googlesource.com/platform/external/conscrypt/+/edf7055461e2d7fa18de5196dca80896a56e3540](https://android.googlesource.com/platform/external/conscrypt/+/edf7055461e2d7fa18de5196dca80896a56e3540)