ONE CLASS TO RULE THEM ALL
0-DAY DESERIALIZATION VULNERABILITIES IN ANDROID

Or Peles, Roee Hay
IBM Security
{orpeles,roeeh}@il.ibm.com

Abstract

We present previously unknown high severity vulnerabilities in Android.

The first is in the Android Platform and Google Play Services. The Platform instance affects Android 4.3-5.1, M (Preview 1) or 55% of Android devices at the time of writing[1]. This vulnerability allows for arbitrary code execution in the context of many apps and services and results in elevation of privileges. In this paper we also demonstrate a Proof-of-Concept exploit against the *Google Nexus 5* device, that achieves code execution inside the highly privileged `system_server` process, and then either replaces an existing arbitrary application on the device with our own malware app or changes the device's *SELinux* policy. For some other devices, we are also able to gain kernel code execution by loading an arbitrary kernel module. We had responsibly disclosed the vulnerability to Android Security Team which tagged it as `CVE-2015-3825` (internally as `ANDROID-21437603/21583894`) and patched Android 4.4 / 5.x / M and Google Play Services.

For the sake of completeness we also made a large-scale experiment over 32,701 of Android applications, finding similar previously unknown deserialization vulnerabilities, identified by `CVE-2015-2000/1/2/3/4/20`, in 6 SDKs affecting multiple apps. We responsibly (privately) contacted the SDKs' vendors or code maintainers so they would provide patches. Further analysis showed that many of the SDKs were vulnerable due to weak code generated by *SWIG*, an interoperability tool that connects C/C++ with a variety of languages, when fed with some bad configuration given by the developer. We therefore worked closely with the *SWIG* team to make sure it would generate more robust code — *patches are available*.

[1] https://developer.android.com/about/dashboards

I. Introduction

Android is the most popular mobile operating system with 78% of the worldwide smartphone sales to end users in Q1 2015 [1].

Android apps are executed in a sandboxed environment to protect both the system and the hosted applications from malware [2]. The Android sandbox relies on the Linux kernel's isolation facilities. While sandboxing is a central security feature, it comes at the expense of interoperability. In many common situations, apps require the ability to interact. For example, the browser app should be capable of launching the Google Play app if the user points toward the Google Play website. To recover interoperability, a key aspect of the Android architecture is Inter-App Communication (IAC) , which enables modular design and reuse of functionality across apps and app components. The Android IAC model is implemented as a message-passing system, where messages are encapsulated by `Intent` objects. Through `Intents`, an app (or app component) can utilize functionality exposed by another app (or app component), e.g. by passing a message to the browser to render content or to a navigation app to display a location and provide directions to it. For implementing IAC, including sending and receiving `Intents`, Android uses a mechanism called *Binder*. As described thoroughly by [3],in the center of the *Binder* mechanism is the *Binder* Linux driver, that transfers messages between processes. Each such Inter-Process Communication (IPC) message is a '*transaction*'. When an application talks to a service, it does so using proxy and stub methods that get translated to *Binder* transactions under the hood, albeit appear like a local function call to the developer.

Due to its significance, much past research has been invested, from multiple angles, in IAC. The research varies from *Testing* [4, 5, 6], *Verification* [7], *Prevention* [8, 9] and *Offensive Security* [10, 11, 12, 13, 14, 15, 16].

II. Serialization

Intents may contain arbitrary data of arbitrary types via a provided Bundle[2] object. For instance, an application may provide another app with a String which can then be accessed via Bundle.getString(String key) or through Intent.getStringExtra(String key). Moreover, a completely arbitrary object can be sent and later be accessed by the recipient via the Bundle.getObject(String key) method. In order to be able to send objects via IAC they must be serialized (or use the Parcelable interface explained briefly below). The recipient needs to deserialize them upon instantiation.

General-purpose serialization of objects can be achieved by implementing the Serializable interface in the object's class. Object members which shall not be serialized (for example, pointers used in native code) can easily be declared by the developer by providing the *transient* modifier before each of such a member. By default, during deserialization, ObjectInputStream's defaultReadObject method is called, receives the class name of the object being deserialized from the stream, and instantiates it, populating its fields that are non-*transient* and non-*static* with the stream's data. In addition, developers can implement special methods in their classes, that will override the default serialization methods. These methods, covered by the Java Documentation[3] include readObject, readResolve and writeReplace.

In addition, Android provides another serialization facility through the Parcleable interface. Objects of implementing classes can be placed inside a Parcel[4] object which is designed as a high-performance IPC transport.

III. The General Android Problem

In 2014, Jann Horn has disclosed a significant vulnerability [17] in Android. The vulnerability, identified as CVE-2014-7911, was a serious flaw in the way Android derserialized objects via ObjectInputStream. In Android versions earlier than 5.0, the deserializing code did not verify that the received object is indeed serializable by making sure that its class implements the Serializable interface. This allowed for inserting arbitrary objects (available to the target's class loader) into a target app or service. Things became more serious because oftentimes the inserted object is automatically deserialized, and later freed by the Garbage Collector (GC), which calls the finalize method of the deserialized object. The finalize method may switch into native code that may access non-*transient* pointers provided by the deserialized object, that the adversary controls. In current Android versions, any object put inside a Bundle can be initialized by a victim app (or service) if the latter only uses (touches) a Bundle arriving from the attacker's IAC. Using a Bundle is a very common behavior among apps.. Internally, when the Bundle is touched (e.g. by calling getStringExtra() or similar), it unparcels itself - initializing (and instantiating) **all** of its values (even unused ones). These values may include Serializable and Parcelable objects, that will get deserialized (or unparcelled). Together with his disclosure, Horn released a Proof-of-Concept (PoC) code crashing Android's system_server which runs under the system context. Horn's PoC worked by providing system_server with an evil, non-Serializable android.os.BinderProxy object, by inserting it into the setApplicationRestrictions's method Bundle parameter. While Horn's PoC only crashed system_server, it clearly demonstrated the problem, which indeed proved to be exploitable as a few months later Yaron Lavi and Nadav Markus released a write-up [18] describing a fully working exploit.

Despite the fact that CVE-2014-7911 has been patched, there can still be Serializable classes available to applications that are dangerous to load if they can be controlled by the adversary. One prominent scenario happens if the developer had forgotten to add the *transient* modifier to a sensitive member (such as a pointer).

This game is asymmetric as one vulnerable class that is available to the default Android class loader is enough for all apps (or one highly-privileged service) to be compromised. Objects of such classes may cause damage in several ways, including automatic code execution of their finalize method with deserialized params or by other methods (such as writeReplace or writeObject) that can be called implicitly .

Hypothetically, even in an ideal world where there aren't such vulnerable classes available in the Android framework, such classes could be found in third-party frameworks (SDKs) or in specific apps - making a more targeted attack.

It is important to note that Bundle isn't the only vector. Each code path that leads to Parcel's readValue, readParcelable or readSerializable will also instantiate whatever object is on the stream (that is loadable).

[2] http://developer.android.com/reference/android/os/Bundle.html

[3] http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html/

[4] http://developer.android.com/reference/android/os/Parcel.html

IV. Finding a Vulnerable Class

We wanted to find a class available to any Android app that both implements the `Serializable` interface, holds a native pointer as a member without declaring it to be *transient*, and also provides a `finalize` method which accesses that pointer. The `finalize` method does not have to be implemented by the class itself, but rather by an ancestor class, accessing the pointer via a call to an overidden method.

We automatically created a list of candidate classes using the following mechanics. We first wrote a small app with a single `Activity`, and then ran it, waiting for a debugger, using the following `am` command, under an emulator running Android 4.2 through 5.1.1.

```
am start -D -n <activity class>
```

We remotely attached our custom debugger to the application through `JDWP`. On top of `JDWP` we ran a Java code that utilizes `JDI` as shown by Figure IV.1. We ignored the `Object`'s and `Enum`'s `finalize` implementations because they were simply empty functions, creating a lot of uninteresting candidates. We printedout only classes which have at least one attacker-*controllable* field. A field is attacker-*controllable* if it is (1) non-*transient* (2) non-*static* (3) all of the classes in the class hierarchy from the candidate class to the class that actually defines the field (can be the candidate class itself) are `Serializable`. (4) of a `Serializable` type (or primitive) (5) no `readObject` / `readResolve` implementation that prevents us from controlling it.

Note that this code does not check for the 5^{th} property of our attacker-controllable definition, so it only yields an upper bound, which we can verify manually. This code yields a couple of classes:

1) `java.util.regex.Pattern`[5]
2) `com.android.org.conscrypt-OpenSSLX509Certificate`[6] in Android 4.4 and `org.apache.harmony.xnet.provider-.jsse.OpenSSLX509Certificate`[7] in Android 4.3.

We then analyzed the candidate classes by hand. The first class has an interesting `finalize` method which accesses a native pointer, `address`, however, marked as *transient*, thus cannot be controlled by the adversary. We had this false positive because this class had another non-*transient* field which was not interesting. We therefore were left with the second candidate only.

[5] `http://androidxref.com/5.1.0_r1/xref/libcore-/luni/src/main/java/java/util/regex/Pattern.java`

[6] `http://androidxref.com/5.1.0_r1/xref/external-/conscrypt/src/main/java/org/conscrypt-/OpenSSLX509Certificate.java`

[7] `http://androidxref.com/4.3_r2.1/xref/libcore-/luni/src/main/java/org/apache/harmony/xnet/provider-/jsse/OpenSSLX509Certificate.java`

```
VirtualMachine vm = <retrieved when attached to
    debuggee>
for (ReferenceType ref : vm.allClasses())
{
  if (!(ref instanceof ClassType))
  {
    continue;
  }
  ClassType cref = (ClassType)ref;

  boolean isSerializable = false;
  for (InterfaceType iref : cref.allInterfaces())
  {
    if ("java.io.Serializable".equals(iref.name()))
    {
      isSerializable = true;
      break;
    }
  }
  if (!isSerializable) continue;

  boolean hasFinalize = false;
  for (Method m : cref.methodsByName("finalize"))
  {
    String declaringClass = m.declaringType().name();
    if (!"java.lang.Object".equals(declaringClass) &&
        !"java.lang.Enum".equals(declaringClass))
    {
      hasFinalize = true;
      break;
    }
  }
  if (!hasFinalize) continue;

  boolean hasField = false;
  for (Field f : cref.fields())
  {
    if (f.isStatic()) continue;
    if (f.isTransient()) continue;

  // isSerializable is our own implementation
    if (isSerializable(f, cref))
    {
      hasField = true;
      break;
    }
  }
  if (!hasField) continue;

  System.out.println(cref.name());
}
```

Figure IV.1. JDI code that finds candidate vulnerable classes

The more pedantic reader would notice that our experiment was limited to the preloaded classes only. In order to overcome that, we created a list of loadable classes by parsing the output of the `oatdump` tool over `boot.art`. This resulted in 13321 loadable classes under our Android 5.1 build. One could then try to add them to `/system/etc/preloaded-classes`, and re-run our experiment. We decided to take a different approach and write a small Android app, that loads the classes using *Java Reflection*, and examines whether they are candidate to be vulnerable, by the same criteria defined above. We did not find any additional candidates.

V. Attacking `OpenSSLX509Certificate`

`OpenSSLX509Certificate` is `Serializable` since it is a descendant of `Certificate` which is `Serializable`. It also has a `finalize` method which calls a native function with its `mContext` member, which is a pointer of type `long`, as an argument. Since `OpenSSLX509Certificate` doesn't implement any special deserialization methods (`readObject` or `readResolve`), and since `mContext` is attacker-*controllable* (as per our definition in Section IV), the adversary can control it. Eventually, as explained in Section III, `OpenSSLX509Certificate`'s `finalize` method will be invoked automatically by the GC. The `finalize` method then invokes `NativeCrypto.X509_free` with `mContext` as a parameter. Its execution ends with a decrement of `*(int *)(mContext + 0x10)`. (In native code.)

Constrained "Write What Where" in user-mode memory can be achieved by decrementing a positive WORD's value one by one (also by accessing its high/low HALFWORDs with unaligned addressing). Freeing of an arbitrary object could also be achieved, but there's no need as we managed to create a fully functional exploit without that.

Note that instead of relying on the GC to call `finalize`, we could have abused the `OpenSSLX509Certificate`'s `writeReplace` method, that is inherited from `Certificate`. As explained in Section III, when implemented in a class, this method overrides the default method for serializing an object. The `writeReplace` method in our case calls `getEncoded`, a method implemented by `OpenSSLX509Certificate` that leads to the invocation of a native function (`NativeCrypto.i2d_X509`) with our attacker-*controlled* parameter (`mContext`). Unlike `finalize`, this method doesn't depend on the GC, but is triggered immediately when our `OpenSSLX509Certificate` object is serialized (at the other end). One way of abusing it would be sending an `Intent` with `OpenSSLX509Certificate` as an `Intent` *Extra* to an application that uses (touches) the `Intent`'s *Extras* `Bundle`, and later serializes the `Bundle` which will trigger a `writeReplace` call on our deserialized object.

VI. Proof-of-Concept Exploit

In order to demonstrate the impact of such a vulnerability, we developed and successfully tested a PoC exploit, running under *Google Nexus 5 Hammerhead* equipped with Android 5.1.1. The exploit can easily be ported to run on other devices running vulnerable Android versions. This paper describes a PoC that demonstrates the feasibility of the attack. It does not include any exploit code. The exploit's fundamental goal is to achieve code execution in an arbitrary app or service. One significant candidate is a process with elevated privileges - Android's `system_server`, which hosts many core services such as Android's *Package Manager*, *Activity Manager* and *Power Manager* services. It runs under UID 1000 - Android's `system` user. We attack `system_server` by inserting our objects into the `setApplicationRestrictions` method's `Bundle` parameter, see [17]. After being able to run code in the system_server process, we demonstrated three different goals the exploit can achieve: The first is replacing an application already on the target device with our own malware app. The second goal we achieved was completely bypassing *SElinux*, by changing and reloading the *SElinux* policy. Third, on devices running kernels that were compiled with loadable modules support (not recent *Google Nexus* devices), we were also able to leverage the exploit to gain arbitrary kernel code execution. This was done by making our shellcode, running inside `system_server`, load our own kernel module – see Section VII-C.

A. Constrained "Write What Where"

As we mentioned in Section V, we are able to decrement the value of a positive integer by one. This is because our pointed object is believed to contain a reference counter of type signed integer at offset 0x10. Thus every time `NativeCrypto.X509_free` is called, it gets decremented and the function returns (without freeing anything). When the value reaches zero, the flow towards `free` continues, and may cause a crash by dereferencing an unmapped pointer.

Because the decrement doesn't crash the remote app (on a positive integer), we can re-run it multiple times, decrementing the target WORD until it reaches the value of 1.

In order to do it efficiently, we include many (0*x*500) `OpenSSLX509Certificate` objects inside each *transaction*'s Bundle, making the WORD's value decrement faster. (Each *Binder* transaction is limited to 1MB in size otherwise a `TransactionTooLargeException` is thrown[8].) In order to further improve the efficiency, to allow us to both target WORDs that initially are negative, and set WORDs with a negative value without crashing, we can also decrement from the middle of our target WORD (2 bytes aligned), lowering the value of the upper half of our WORD to reach a desired value, or until our WORD's MSB (sign-bit) is zeroed, so that we will later be able to operate on the target WORD

[8]http://developer.android.com/reference/android/os/-TransactionTooLargeException.html

directly and also decrement the lower HALFWORD. In case the desired value is negative, after setting the lower HALFWORD, we can get back to the $address+2$ WORD and set the higher HALFWORD to a negative value. The ability to do this of course depends, again, on the value of the WORD we decrement and of the next adjacent HALFWORD.

Note that for the constrained "Write What Where" we have to know the original value of the target WORD in the remote process as well. This is possible thanks to the Zygote creation model, *see* Section VI-D.

B. Controlling the Program Counter (PC)

Now that we have the ability to control a value from another process's address space, we can use it to subvert a target program's logic, by changing authentication or privilege states for example. In addition, we can try to achieve arbitrary code execution. Android, a Linux-based OS, has memory page permissions. Since we didn't find any memory page that is both writable and executable, we thought we could modify a function's address from the Global Offset Table (`.got` section). Unfortunately for us, the shared libraries in memory were all compiled with the Relocation Read Only *(RELRO)* flag. This compile-time flag instructs the loader to resolve all symbols' addresses immediately during the binary loading, and then set their memory pages to read-only, making `.got` overwrite impossible.

C. Writable Function Pointers

Another approach for gaining code execution by controlling memory is by changing a writable function pointer. In some cases, libraries let other apps hand pointers of callback functions, which are invoked in specific places in the library's code. For example, libraries may allow an application to hand a callback for logging purposes, that will be invoked every time the library needs to write to a log. These callbacks sometimes appear as global function pointers in the library's (writable) *data* section, making them a perfect target for achieving arbitrary code execution.

In our case, we found that *libcrypto* (the OpenSSL library) offers this functionality which is used by *libjavacrypto* (conscrypt's native library). *libcrypto* holds the pointers as global variables in the *data* section, and thus they are writable. We found calls to the callback throughout the code.

Specifically, we chose to override the `id_callback` function pointer to point to our code. We then make the target process reach a code path that invokes the callback function. We do this again using the same trick of serializing an object that will be instantiated at the other end. This time we put an `OpenSSLECPrivateKey` object, with bad contents in its byte array member inside the bundle. The byte array won't be successfully decoded as PKCS8, but will help us in later stage, see Section VI-H. `OpenSSLECPrivateKey` implements the `readObject` method, that overrides the default deserialization method, and thus is called during object deserialization. This function calls the `NativeCrypto.d2i_PKCS8_PRIV_KEY_INFO` function with the byte array we supplied. Our supplied byte array makes the function call the `throwExceptionIfNecessary` function (after failing to parse our byte array to a valid value), that eventually leads to an invocation of `CRYPTO_THREADID_current`, that directly calls our callback function .

A minor drawback we encountered of using the `id_callback` function pointer is that its adjacent HALFWORD is zeroed. This means that by decrementing this function pointer, we are practically limited to positive addresses $[1, 0x7fffffff]$. We practically bypass this limit in Section VI-G.

D. Bypassing Address Space Layout Randomization (ASLR)

ASLR is a security mechanism whose purpose is to make it harder for attackers to build a reliable exploit. ASLR randomly arranges key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. ASLR could have easily prevented us from gaining code execution by overriding a specific function pointer, because we wouldn't have known where the function pointer lies in memory, and not less important, what its initial value is, which is an address of a function in the `libjavacrypto` module.

Android's Zygote Process Creation Model: In Android, every application, and most of the service processes are created by forking from a single process named *Zygote*. The design was intended to improve the responsiveness of applications at launch-time, but it adversely affects the effectiveness of ASLR. The Zygote process is created among the first processes at boot time, and contains a full Dalvik or ART Virtual Machine instance with frequently used classes preloaded. Lee et al. [19] concluded that "all running apps inherit the commonly used libraries from the Zygote process and thus share the same virtual memory mappings of these libraries, which represent a total of 27 MB of executable code in memory". Other research also abused Zygote. For example, Kaplan et al. [13] leaked a random value generated in early boot in order to reconstruct the Linux Psuedo-Random Number Generator (PRNG) state.

Thanks to Zygote, by observing our own, malicious process's, memory we are able to realize the function pointer's address and value in the remote target process's memory. Both of these are addresses in libraries

that were forked from Zygote (even in `system_server`, which also forks from Zygote).

E. Towards Code Execution

Currently, using the function callback we decremented, we have the ability to control the PC, i.e. to execute code from an arbitrary address (constrained to a positive int value) in memory. As we don't currently have code instructions that will perform the tasks we want in the target process's memory, at this stage we manipulate memory and register values in order to prepare our own shellcode for execution. We then jump to it. We achieved this by passing the shellcode bytes to the target process, and by chaining multiple Return-Oriented Programming (ROP) gadgets as explained in the next sections. In order to execute the chain of gadgets using returns, we placed the gadgets' addresses on the (pivoted) stack and jumped to the first gadget. We used the *Stack Pivot* technique to achieve this (see Section VI-H).

F. Reliably Locating our Shellcode in the Remote Process

For the *Stack Pivot* and later steps, we need to have a memory buffer (for the shellcode) at a reliable location in the remote process's memory. Traditionally, placing the shellcode in the target process's memory is done by using a technique called *Heap Spraying*, that basically involves filling the target process's heap memory with repetitions of the desired shellcode, increasing the probability that the shellcode will reside at an arbitrary address during the exploitation. In our case, though, the byte array that we fully control during the deserialization of the `OpenSSLECPrivateKey` object for triggering our code execution (see Section VI-C) happens to be directly pointed by the `fp` (`r13`) register at the time our first gadget's (see G1, Section VI-J) opcode is reached.

G. Running Code from an Unconstrained Arbitrary Address

The next adjacent HALFWORD of `id_callback` happens to be zero, meaning the value we can overwrite it with is limited. Due to the fact that `id_callback`'s value always happens to be above *0x7fffffff* (it is set to the address of a function under `libjavacrypto`, which sits at high memory), we can practically change `id_callback`'s value to any positive integer. To overcome this limitation, and gain the ability to execute gadget at any user-space address, we wanted to find a gadget residing at a positive integer address, that will let us jump to any address.. Observing the memory layout of Android processes forked from *Zygote*, we noticed `boot.oat` (used by ART) has a large executable mapping around the 0x70000000 area. Since our controlled memory is pointed by `fp` (see Section VI-F) we searched for a gadget that directly jumps to a dereferenced value relative to `fp`. We haven't found such a gadget, but instead, found a gadget (G1, see Section VI-J) that takes `fp[0]`, dereferences it 2 more times and jumps to the resulting address. Since we control `fp[0]`'s value (the very first WORD in our byte array), and we could make sure that the dereferenced pointer chain will all be at addresses whose adjacent HALFWORDs are a positive short, allowing us to fully control the jump address by our constrained decrement. We thus have to set 2 additional WORDs using the decrementation method described in Section VI-A before running the gadget.

H. Stack Pivoting

A common method for attackers to conveniently control program execution involves pointing the stack pointer to an attacker-controlled memory. This allows the attacker to prepare a call stack that includes the addresses of the desired ROP gadgets so that each time a return instruction (`pop {...,pc}`) occurs, the next gadget runs. The second gadget we run (after achieving unconstrained arbitrary address execution) is G2 (Section VI-J). It changes the value of `sp` to a value relative to `fp`, and then returns. Thus from that point forwards, our controlled byte array (from Section VI-C.) becomes the program's stack.

I. Running Our Shellcode Despite SElinux

Security-Enhanced Linux (SElinux) is a security enhancement to Linux which allows fine-grained control over access control. It is comprised of kernel modifications and user-space tools that have been added to various Linux distributions. It was incorporated into Android in Android 4.3, in *permissive* mode (permission denials are logged but not enforced), and gradually moved to full enforcement in Android 5.0 [20]. *SElinux* assigns processes with a security domain (or type), adding the ability to restrict specific processes' from doing certain actions (regardless of the user running them). The types' restrictions are defined in `.te` files. *SElinux* takes a secure white-list approach where everything is disallowed, except for the specific rules defined for the process.

We decided to run our own shellcode inside the `system_server` process due to the amount of *SELinux* capabilities this process has. (Which is due to the fact that it hosts many system services.) Our shellcode can then do many sensitive actions. The process is

subject to the restrictions of the system_server type [9]. A required step for running our shellcode, which we put as part of the OpenSSLECPrivateKey object's byte array, is to map it as an executable memory. By default, such an action will be restricted because of *SElinux*, but as one can see, system_server.te includes the "allow system_server self:process execmem" rule, giving the system_server process the PROCESS__EXECMEM permission, which allows it to map anonymous memory as executable. We take advantage of this fact, and use our ROP chain (explained in the next section) to map a readable, writable and executable (rwx) memory location, copy our shellcode to that location, and run it from there.

We note that we had to overcome a caching problem. ARM has two separate memory caches types - one for instructions and another for data [21]. When we first copied our code to the rwx memory area, it was written to the data cache, and was not immediately flushed into memory. Thus when we jumped to the shellcode location, the processor fetched instructions for execution directly from memory, while our shellcode was still residing in the data cache. In order to overcome this, we copy a large memory amount after our shellcode, filling the data cache. By doing so, we manage to evict (flush) our shellcode from the data cache to memory.

J. ROP Chain

The series of of ROP gadgets we use is summarized here. The gadgets map a rwx memory area in (the unused) address 0x50000000, copy the shellcode there and jump to it. All the gadgets were found inside libraries that are loaded by Zygote. They were found either manually (in boot.oat, which isn't a valid ELF binary), or by using ROPGadget[10].

(G1) *Jump to an Arbitrary Address*: `mov r1, fp; ldr r0, [r1, #0]; ldr.w r0, [r0, #444]; ldr.w lr, [r0, #44]; blx lr`. Found in boot.oat, which is at a positive integer address . This gadget is used for jumping to an unconstrained arbitrary memory address by taking a value relative to fp and jumping to it.

(G2) *Stack Pivot*: `mov sp, fp`, and return.

(G3) *Allocate Buffer on Stack*: `add.w sp, sp, #0x420`, and return. Used to allow more stack space before calling libc functions. It is needed since our byte array happens to be near the beginning of a page, and we don't want the stack to grow down to an unmapped memory area.

(G4) *Map RWX memory*: `pop {r0, r1, r2, r3, pc}`. This gadget is used for running `mmap(0x50000000, shellcode_size + EXTRA_FOR_CACHE_EVICTION, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, 0, 0)`. Note that mmap's prologue includes pushing the LR register to the stack. In its epilogue it is popped from the stack and jumped into. Because we couldn't easily find a gadget that sets LR value, we used a small trick: we don't jump to mmap's real beginning, but rather into mmap just after the instruction that pushes LR. This way LR isn't placed on the stack at all, and thus later in the epilogue, when mmap pops a value from the stack (which was meant to be the saved LR), it gets what we placed on the stack in advance, which is the address of the next gadget.

(G5) *Progress the Stack*: `pop {r0, r1, r2, r3, pc}`. This gadget advances the stack pointer abit forward before jumping to next gadget, so we don't have an overlapping with the previous gadget's mmap 5^{th} and 6^{th} parameters, which lie on the stack.

(G6) *Call memcpy and return to Shellcode*: `add r1, sp, #0x24c; pop {r0, r2, r5, r7, pc}`. This gadget prepares r1 (memcpy's first argument) to point to our shellcode in memory, and then calls `memcpy(0x50000000, OUR_SHELLCODE, shellcode_size + EXTRA_FOR_CACHE_EVICTION)`. Note that we used the same trick as mentioned in the mmap gadget, for setting the return address to be our shellcode location.

K. The Shellcode

We compiled a shellcode written in C that uses library functions to perform its actions. The shellcode is compiled with placeholders for the library functions, that are resolved (using the Zygote method, *see* Section VI-D.) and changed by our application before triggering the exploit (i.e. before the shellcode is sent). In Section VII we list some options for the shellcode.

VII. Impact

In this section we list a few options for a shellcode targeting system_server. While this list is incomplete, it surely demonstrates the severity of the vulnerability.

A. Replacing Existing Applications

The shellcode can replace any user app on the device (residing in /data/app/<target-app>/base.apk) with arbitrary (malicious) app. It can then reboot the device (so changes will take place immediately) by setting the sys.powerctl property (by calling the libcutil's property_set function). The malware will have access to the original app's data and will be launched instead of the original app.

[9] https://android.googlesource.com/platform/external/sepolicy/+/android-5.1.1_r6/system_server.te

[10] https://github.com/JonathanSalwan/ROPgadget

B. Completely Bypassing SElinux

On Android, the *SElinux* policy can be updated by writing the policy configuration files to the `/data/security/current` directory, and setting the `selinux.reload_policy` property to 1. In the original *SElinux* policy that comes with Android 5.1.1 (and others), the only process that is allowed to write to files under the `/data/security` is the process we control, `system_server`. One can find this allowance in the `system_server.te` file under the macro `selinux_manage_policy(system_server)` which gets transformed (definition in the *te_macros* file) to allow permissions for directory, file or link creation inside `/data/security`, and permission to set the `selinux.reload_policy` property.

In order to change the policy, we need to create the `current` directory inside `/data/security`, and write the policy files inside it. The policy consists of the `sepolicy` binary file, and 6 textual files[11]: (1) `selinux_version`. (2) `file_contexts`. (3) `property_contexts`. (4) `seapp_contexts`. (5) `service_contexts` and (6) `mac_permissions.xml`. In order to load the new policy, the contents of `/data/security/current/selinux_version` file must be equal to the contents of the `/selinux_version` file. This is not a difficult task, since the latter just contains the current Android `BUILD_FINGERPRINT` value which can be retreived by reading the `ro.build.fingerprint` property using `libcutils' property_get`. As for the `mac_permissions.xml` file, we create a symbolic link to the original file in `/system/etc/security/mac_permissions.xml`. We simply create a symbolic link, because in contrast to the rest of the files, this file isn't opened with the `O_NOFOLLOW` flag, and thus can follow links. As for the contents of the rest of the files, we use files generated in advance by either building a custom Android AOSP image (after changing the *SELinux* rules in the textual configuration as we like), or take the original files of the same Android build, but edit the binary `sepolicy` file using the `sepolicy-inject` tool [22]. By taking the second approach, we ran our exploit and successfully changed sepolicy domains (`system_server` and `shell_exec`) to be *permissive*. We then ran a shell command `execve` with `sh`, that we were not allowed to perform previously.

C. Kernel Code Execution

Linux kernels can be compiled with loadable modules support, by specifying the `CONFIG_MODULES` option. When this feature is supported, privileged processes can load arbitrary modules into the kernel using a syscall, as long as the kernel was not compiled with module signature verification (`CONFIG_MODULE_SIG`). As can be seen in the *SElinux* policy rules for the `system_server` process (`system_server.te`, explained in Section VI-I), `system_server` is not restricted from loading kernel modules, thanks to the "`allow system_server kernel:system module_request`" rule.

For security reasons, many Android kernels, including Google Nexus 5's were not compiled with `CONFIG_MODULES`, and hence this section is irrelevant to them. Despite that, we have encountered other new popular up-to-date Android devices whose kernels do support `CONFIG_MODULES` (without `CONFIG_MODULE_SIG`).

For testing this basic idea (on our device), we conducted the following experiment: we replaced our Nexus 5 device's kernel with a kernel we compiled from AOSP with `CONFIG_MODULES` support. We then used our exploit against it, with a shellcode that loads our kernel module (that we compiled in advance) into memory. We were indeed able to verify the exploit's success by seeing our module's `printk`'s messages appear in the device's `dmesg` (kernel messages) log.

VIII. Finding Vulnerable Classes in SDKs and Apps

As mentioned in Section III, vulnerable classes can be found in specific apps or frameworks, implying a more restricted (targeted) attack. We therefore decided to analyze the 32,701 most downloaded and free Android apps in order to find such classes. Since using our aforementioned runtime technique to conduct this experiment would take hours to complete, we decided to use a different approach. We created a tool that runs `dexlib2`[12] over the apps' dex files. `dexlib2` gives a *Reflection*-like interface which allowed us to statically find classes that meet the criteria defined in section IV (again without property 5) in mere 93 minutes. One caveat of this approach is that `dexlib2` is limited to the classes defined in the given APK's dex files, so our approach would have false negatives on classes which inherit from a `Serializable` Android SDK class or from an Android SDK class that has a `finalize` method. In order to compensate for that we created two caches. The first cache is of `Serializable` Android SDK classes with their attacker-*controllable* fields (see Section IV), while the second is of SDK classes that contain a `finalize` method. Both are based on classes found in `boot.art` of our Android 5.1 build (using the technique explained in Section IV). We found a total of

[11] http://selinuxproject.org/page/NB_SEforAndroid_1-#Device_Policy_File_Locations

[12] `https://github.com/JesusFreke/smali/tree/master/dexlib2`

358 classes over 176 APKs that met our criteria. Our analysis had quite a few interesting findings.

A. Vulnerable Classes Generated by SWIG (CVE-2015-2000/1/3/4/20)

We witnessed several APKs with a `Serializable` implementing class that contained a non-*transient* long member named `swigCPtr`. This immediately raised a red flag. Investigating further revealed a very similar to `OpenSSLX509Certificate` situation, with the `swigCPtr` variable passed to native code.

SWIG[13] or 'Simplified Wrapper and Interface Generator' is an interoperability tool that connects C/C++ code with a variety of high-level languages including Java. Running *SWIG* over a C++ class generates 2 additional components: A Java wrapper to be used by Java apps, and a JNI interface that interconnects the wrapper to the original C++ code. For example, running SWIG over the very simple C++ code shown by Figure VIII.1 generates the Java wrapper class found in Figure VIII.2 and the JNI binding code partially given by Figure VIII.3.

```
class Foo
{
public:
  Foo() {}
  virtual ~Foo() {}
};
```

Figure VIII.1. C++ code as an input to SWIG

```
public class Foo {
  private long swigCPtr;
  protected boolean swigCMemOwn;
  ...
  protected void finalize() {
    delete();
  }
  public synchronized void delete() {
    if (swigCPtr != 0) {
      if (swigCMemOwn) {
      swigCMemOwn = false;
      exampleJNI.delete_Foo(swigCPtr);
    }
    swigCPtr = 0;
    }
    super.delete();
  }
  ...
}
```

Figure VIII.2. SWIG-generated Java wrapping class

As shown by Figure VIII.5, it is possible to instruct *SWIG* to make the wrapping class extend another or implement a specific interface by having a special *SWIG* interface file (Figure VIII.4) .

[13]http://www.swig.org/

```
SWIGEXPORT void JNICALL Java_exampleJNI_delete_1Foo(
    JNIEnv *jenv, jclass jcls, jlong jarg1) {
  Foo *arg1 = (Foo *) 0 ;
  (void)jenv;
  (void)jcls;
  arg1 = *(Foo **)&jarg1;
  delete arg1;
}
```

Figure VIII.3. Part of the JNI binding code

```
%typemap(javabase) Foo "Bar";
%typemap(javainterfaces) Foo "Baz"
```

Figure VIII.4. SWIG interface with custom class extension and interface implementation

This means that if the developer explicitly chose to implement the `Serializable` interface, by using *SWIG*'s `javainterfaces` typemap, or more commonly, implicitly and probably accidentally by extending a class, using *SWIG*'s `javabase` typemap, which is `Serializable`, the generated code would be vulnerable to our attack since the generated `swigCPtr` would not be *transient*. It should be noted that the generated native code (Figure VIII.6) for a class with a virtual destructor can be exploited more easily since the attacker can trivially control the PC due to the fact that its address is fetched from the object's provided *vtable* whose pointer (`swigCPtr`) the attacker fully controls. Therefore it does not require overwriting a callback pointer as opposed to the technique shown in Section VI. As for non-virtual destructors, the flow reaches the global `delete` operator which reduces the problem to exploiting an arbitrary `free`.

The fact that *SWIG* can generate vulnerable code has an amplification potential. We indeed found multiple SDKs that included vulnerable *SWIG*-generated classes which further amplified the problem. Among the vulnerable SDKs were *Jumio* (CVE-2015-2000), *MetaIO* SDK (CVE-2015-2001), *PJSIP* PJSUA2 (CVE-2015-2003), *GraceNote* GNSDK (CVE-2015-2004) and *MyScript* (CVE-2015-2020). A total of 18 APKs in our sample set used these SDKs and ended-up to be vulnerable.

We argue that the fact that the application can accidentally implement the `Serializable` interface, by extending a `Serializable` class such as `Exception`, is the root cause of this issue, although we also encountered an explicit implementation of the `Serializable` interface.

B. ArcGis Runtime SDK for Android (CVE-2015-2002)

Similarly, ArcGis Runtime SDK for Android by *esri* contained a class with a attacker-*controllable* pointer which propagated to native code.

```
public class Foo extends Bar implements Baz {
    private long swigCPtr;
    protected boolean swigCMemOwn;
    ...
}
```

Figure VIII.5. Result of using the custom SWIG interface file

```
Java_exampleJNI_delete_1Foo

.text:000023A4 PUSH {R0–R2,LR} es
.text:000023A6 MOVS R0, R2
.text:000023A8 STR R2, [SP,#0x10+var_10]
.text:000023AA STR R3, [SP,#0x10+var_C]
.text:000023AC CMP R2, #0
.text:000023AE BEQ locret_23B6
.text:000023B0 LDR R3, [R2]
.text:000023B2 LDR R3, [R3,#4]
.text:000023B4 BLX R3
.text:000023B6 POP {R0–R2,PC}
```

Figure VIII.6. Compiled JNI code using the Android NDK tool-chain

C. Google Play Services (ANDROID-2153894)

We also encountered the vulnerable `OpenSSLX509Certificate` class in the *Google Play Services (GMS)* APK. Google has tracked it with the above internal identifier and the same CVE as of the Android Platform instance.

IX. Mitigation

We privately disclosed the vulnerabilities to the various vendors and code maintainers prior to the publication of this paper. The disclosure timeline is summarized by Table I. The following describes the various patches that the vendors made to their vulnerable code.

A. Patched `OpenSSLX509Certificate`

Google has fixed the two `OpenSSLX509Certificate` instances by adding the *transient* modifier to the `mContext` member. Google has also backported the patch to Android 4.4 (commit id `0b9d6334acde7460502face82417de40e438a3f4`), 5.0 and 5.1 (commit id `de55e62f6c7ecd57d0-a91f2b497885c3bdc661d3`). The patch is also available in Android M (build `MPZ79M`).

B. Patched SDKs

We reported the issues to the relevant vendors or code maintainers. *Jumio* removed the vulnerable SWIG classes. *esri* has patched the ArcGis Runtime SDK by adding the *transient* modifier to the native pointer. *MyScript* and GraceNote fixed their SDKs by adding the *transient* modifier to the sensitive variables, *PJSIP* patched PJSUA2 by by overriding the `readObject` / `writeObject` methods, effectively making the vulnerable class non-`Serializable`. The MetaIO SDK was fixed as well, as it no longer implements the `Serializable` interface.

C. Patched SWIG

Since the generated vulnerable code was due to bad configuration given by the developer, we do not consider *SWIG* to be vulnerable. This is somewhat analogous to blaming a compiler for buffer overflows. However, even the most competent developers could miss the fact that they accidentallly extended a `Serializable` class. Therefore we decided to contact *SWIG* team which released a more secure version. The patched version generates the `swigCPtr` member with the *transient* modifier.

X. Discussion

While the patches fixed the specific instances that we had found, we feel that a general problem deserves a general mitigation, reducing the impact of such serialization attacks. Since `Bundles` are very common in Android's Inter-Process Communication, we suggest changing the `Bundle`'s default behavior that automatically instantiates all of its values (under `BaseBundle.unparcel`, that is invoked by any 'touch' of the `Bundle`) to a lazy approach, i.e. retrieving only the values of keys it is asked for. Of course by design the problem will still remain, but will depend more on specific developer's code, so less apps will be vulnerable if another vulnerable class is found, significantly narrowing the attack surface. In addition, further protection that prevents arbitrary `readObject`, `readReplace`, `writeObject` or `writeResolve` methods to be called can be achieved by deprecating the current `getParcelable` and `getSerializable` methods (of both `Bundle` and `Parcel`) and change the APIs to include the class name of the expected objects.

Furthermore, mitigating exploitation is also possible. First, as explained in Section VI, ASLR is pretty much useless in Android if the attacker can launch malware. [19] provides a secure replacement for the insecure Zygote implementation. In addition, as our PoC exploit overwrites a function pointer, pointer integrity mechanisms are also possible. One might think that callback pointers must be writable, because of their functionality, i.e. they have to be dynamically changed throughout program lifetime. However, this does not have to be the case. One mechanism that solves this problem is keeping the memory area that holds the function pointers to read only, and the memory will become writable by the callback setter functions just for the period of changing

Product	Identifiers	Report date	Patched version(s)
Android	CVE-2015-3825 ANDROID-21437603	05/22/2015	Android M / 5.x / 4.4
Google Play Services	CVE-2015-3825 ANDROID-21583894	06/01/2015	7.5.73
Jumio SDK	CVE-2015-2000	06/11/2015	1.5.0
MetaIO SDK	CVE-2015-2001	06/11/2015	6.0.2.1
esri ArcGis	CVE-2015-2002	06/11/2015	10.2.6-2
MyScript	CVE-2015-2020	06/16/2015	1.3
PJSIP PJSUA2	CVE-2015-2003	06/14/2015	SVN Changeset 5132
GraceNote GNSDK	CVE-2015-2004	06/14/2015	1.1.7
SWIG	-	06/11/2015	3.0.7

Table I
DISCLOSURE LOG

the values. A different mitigation approach, that is discussed by Stefan Rattger [23] proposes to white-list function pointer values to identify cases where the function pointers are being abused. Another interesting mitigation is treating fields as *transient* by default (i.e. an opt-in approach as opposed to the current opt-out one). As for the specific payloads we show in Section VII, disabling `CONFIG_MODULES`,or at least enabling `CONFIG_MODULE_SIG` as well, is a good practice. A more restrictive *SELinux* non-bypassable policy should also be deployed.

In Android 5.0, the `WebView` component was moved to an updatable APK [24], decoupling it from the rest of the system. This had quite a few significant advantages with respect to security. First, it allowed Google to release security patches much faster. Second, it tackled the Android fragmentation problem as patches could be backported for old Android versions more easily, and delivered to various devices all at once. We encourage Google to continue their efforts towards decoupling the vendors' dependent code from the rest of the system, so patches will be available much faster. Our case showed the significance of such a separation, as the Google Play Services instance of `OpenSSLX509Certificate` was updated, in multiple Android versions, 3 days after our report.

As opposed to vulnerabilities found in final products, such as operating systems or applications, where an automatic update mechanism is usually available, the situation is by far worse for SDKs. One vulnerable SDK can affect dozens of apps whose developers are usually unaware of it, taking months to update. For example, a recent study [15] shows that a high severity vulnerability [11] found in Apache Cordova for Android, although been patched for months, still affects dozens of Android apps which use Cordova. The situation is most frustrating for apps which use orphan SDKs, ones that no longer receive security updates. Developers should be aware that depending on 3rd-party SDKs has that significant risk, prepare for alternatives, or choose their used SDKs wisely. Developers also deserve better tools - Gradle and its 3rd-party plugin, Gradle Versions Plugin, which notify the developer when a new version is available for a used SDK, are a pretty good start.

XI. Acknowledgements

We would like to thank the following teams and individuals:

- *Google* and the various SDK teams (*Jumio* / *esri* / *MyScript* / *MetaIO* / *PJSIP* / *GraceNote*) for the prompt responses and patching. This shows without doubt their commitment for security.
- The SWIG team, and especially *William S. Fulton*, for making SWIG more secure.
- *Sagi Kedmi* of IBM X-Force Application Security Research Team for his help with the 32k apps experiment.

References

[1] IDC. Smartphone OS Market Share, Q1 2015, 2015. URL http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, January 2009. ISSN 1540-7993. doi: 10.1109/MSP.2009.26. URL http://dx.doi.org/10.1109/MSP.2009.26.

[3] Thorsten Schreiber. Android Binder: Android Interprocess Communication, October 2011. URL https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf.

[4] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 531–536, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2800-5. doi:

10.1145/2590296.2590316. URL http://doi.acm.org/10.1145/2590296.2590316.

[5] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proc. ISSTA*, pages 118–128. ACM, 2015. URL http://researcher.ibm.com/researcher/files/us-otripp/issta15.pdf.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0643-0. doi: 10.1145/1999995.2000018. URL http://doi.acm.org/10.1145/1999995.2000018.

[7] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382223. URL http://doi.acm.org/10.1145/2382196.2382223.

[8] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing Attack Surfaces for Intra-application Communication in Android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 69–80, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1666-8. doi: 10.1145/2381934.2381948. URL http://doi.acm.org/10.1145/2381934.2381948.

[9] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[10] Roee Hay. Overtaking Firefox Profiles, March 2014. URL http://slidesha.re/1gqiyD3.

[11] David Kaplan and Roee Hay. Remote Exploitation of the Cordova Framework. Technical report, 2014. URL http://www.slideshare.net/ibmsecurity/remote-exploitation-of-the-cordova-framework.

[12] Roee Hay. Android Collapses into Fragments, December 2013. URL https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf.

[13] David Kaplan, Sagi Kedmi, Roee Hay, and Avi Dayan. Attacking the Linux PRNG On Android. In *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014.*, 2014. URL https://www.usenix.org/conference/woot14/workshop-program/presentation/kaplan.

[14] Roee Hay and Or Peles. Remote Exploitation of the Dropbox SDK for Android, March 2015. URL http://www.slideshare.net/ibmsecurity/remote-exploitation-of-the-dropbox-sdk-for-android.

[15] David Kaplan and Roee Hay. IBM X-Force Threat Intelligence Quarterly, 1Q 2015. Technical report, March 2015. URL http://ibm.co/1wEMKV3.

[16] Takeshi Terada. Attacking Android browsers via intent scheme URLs. 2014. URL http://www.mbsd.jp/Whitepaper/IntentScheme.pdf.

[17] Jann Horn. CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream, November 2014. URL http://seclists.org/fulldisclosure/2014/Nov/51.

[18] Yaron Lavi and Nadav Markus. CVE-2014-7911 - A Deep Dive Analysis of Android System Service Vulnerability and Exploitation, January 2015. URL http://bit.ly/1Q7pdzT.

[19] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 424–439, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.34. URL http://dx.doi.org/10.1109/SP.2014.34.

[20] Android. Security-Enhanced Linux in Android. URL http://source.android.com/devices/tech/security/selinux/.

[21] Jacob Bramley. Caches and Self-Modifying Code, February 2010. URL http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code.

[22] Michal Krenek. setools-android with sepolicy-inject, December 2014. URL http://forum.xda-developers.com/android/software/setools-android-sepolicy-inject-t2977563.

[23] Stephen Rattger. PoC: Function Pointer Protection in C Programs, August 2013. URL http://gcc.gnu.org/ml/gcc/2013-08/msg00224.html/.

[24] TJ VanToll. What Android 5.0's Auto-Updating WebView Means for Mobile Apps, November 2014. URL http://bit.ly/1DqQ4RC.