Google

Your Move

Vulnerability Exploitation and Mitigation in Android

Dan Austin

April 2017

Agenda

Introduction

SELinux

Stagefright-inspired mitigations

Reducing attack surface: IOCTL filtering

User-kernel copy mitigations

Complicating kernel to user execution: PAN emulation

Mitigations: Where are they?

$ whoami

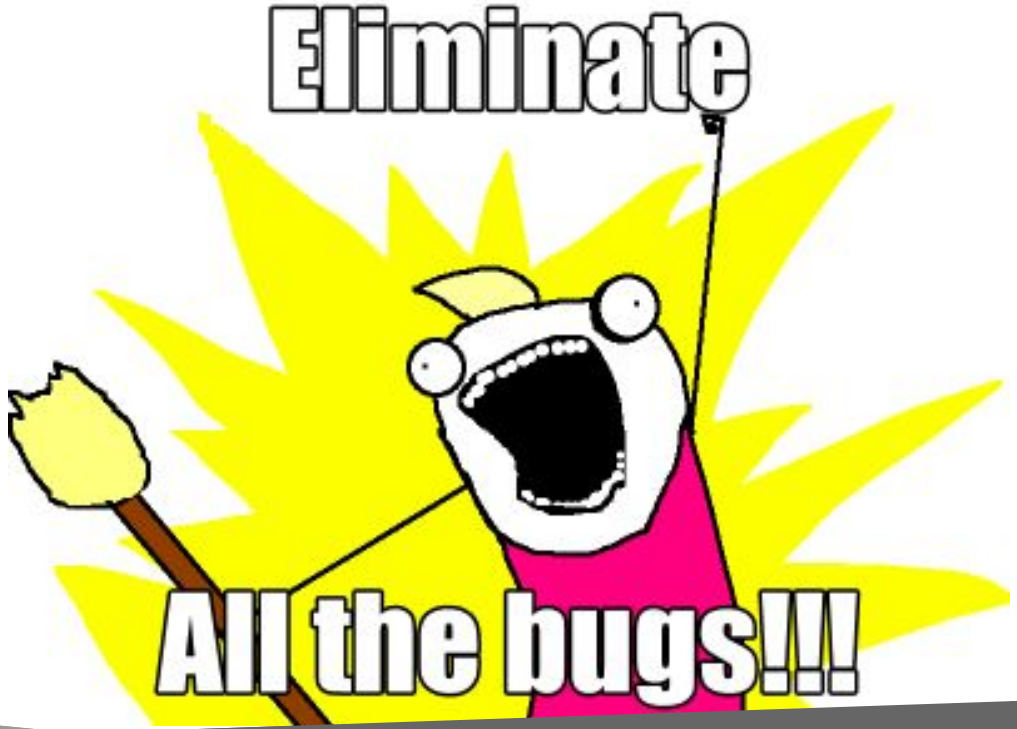
- Dan Austin
- With Google since August 2015
- Android Security
  - “Rehabilitated” Attacker
  - I was cured, alright...

Bug-free code is hard

99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.

127 little bugs in the code...

We could try to eliminate all bugs



Or, we can make exploitation harder...

SELinux

SELinux

- Mandatory Access Control (MAC) System
- One Central Security Policy
- Kernel Enforced
- Policy is fine grained / label based
- Orthogonal to UIDs or GIDs (even root)
- Default Deny (Principle of Least Privilege)
- Works in conjunction with the standard Unix DAC model

Neat! So, what does this mean?

It means....

- CVE-2017-0583: `cpaccess` module gives userspace direct control over `EL1` (kernel-level) system control registers
  - See `arch/arm64/include/asm/sysreg.h` for register encodings
- Allows for commands to be written to `/sys/devices/cpaccess/cpaccess0/cp_rw`
- Execute `MSR` on the specified CPU by sending the string `s:[op0]:[op1]:[CRn]:[CRm]:[op2]:w:[value]:[cpu]`
- Execute `MRS` on the specified CPU by sending the string `s:[op0]:[op1]:[CRn]:[CRm]:[op2]:r:[unused]:[cpu]`
- Thankfully...

It means.... (translated through SELinux)

```
pixel-xl:/ # ls -lZ /sys/devices/cpaccess/cpaccess0/cp_rw
-rw-r--r-- 1 root root u:object_r:sysfs:s0 4096 1970-05-24 11:31 /sys/devices/cpaccess/cpaccess0/cp_rw

$ sesearch -A -t sysfs -c file,chr_file -p write sepolicy
allow dumpstate sysfs:file { read lock getattr write ioctl open append };
allow gpsd sysfs:file { read lock getattr write ioctl open append };
allow healthd sysfs:file { read lock getattr write ioctl open };
allow init sysfs_type:file { write lock open append relabelto };
allow init_power sysfs:file { read lock getattr write relabelfrom ioctl open append };
allow netd sysfs:file { read lock getattr write ioctl open };
allow nfc sysfs:file { read lock getattr write ioctl open };
allow perfd sysfs:file write;
allow system_server sysfs:file { read lock getattr write ioctl open append };
allow ueventd sysfs:file { read lock getattr write ioctl open append };
allow vold sysfs:file { read lock getattr write ioctl open append };
```

Need to be root *and* a privileged process to take advantage of this

Or in general....

1. Exploit the target userland component
2. Realize SELinux is preventing you from doing anything interesting
3. Exploit the kernel
4. Disable SELinux

Good SELinux policies prevent lateral movement on the system!

Stagefright Inspired Mitigations

What is it?

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```

This can overflow!

And cause corruption here

```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R7, [SP,#0x30]
LDREQ           R7, [SP,#0x30]
LDR             R6, [SP,#0x28]
ADDS            R0, R7, R6
BLX             _Znaj   ; operator new[](uint)
MOV             R8, R0
CBZ             R7, loc_7E6A6
LDR             R1, [SP,#0x40]
MOV             R0, R8
MOV             R2, R7
BLX             __aeabi_memcpy
```

We fixed it!

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    if (SIZE_MAX - chunk_size <= size) {
      return ERROR_MALFORMED;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```

Checking for overflow before use fixes the problem, right?

```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R6, [SP,#0x30]
LDREQ           R6, [SP,#0x30]
LDRD.W          R7, R0, [SP,#0x28]
MOVS            R2, #0
MVNS            R1, R7
CMP             R1, R6
MOV.W           R1, #0
IT LS
MOVLS           R1, #1
CMN             R2, R0
ITT EQ
MOVEQ           R2, #1
MOVEQ           R2, R1
CMP             R2, #0
BNE.W           return_ERROR_MALFORMED
ADDS            R0, R6, R7
BLX             _Znaj   ; operator new[](uint)
MOV             R8, R0
CBZ             R6, loc_7E6BC
LDR             R1, [SP,#0x40]
MOV             R0, R8
MOV             R2, R6
BLX             __aeabi_memcpy
```

NOW we fixed it! (Thanks UBSAN!)

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    if (SIZE_MAX - chunk_size <= size) {
      return ERROR_MALFORMED;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```

All overflows trigger an abort now, exploit is prevented!

```
                BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
                CBNZ            R0, loc_81F2A
                STR             R5, [SP,#0x38]

loc_81F2A                                       ; CODE XREF: .text:00081F26↑j
                LDR             R1, [SP,#0xF4]
                CMN             R5, R1
                BNE.W           call_abort
                LDR             R5, [SP,#0xF0]
                NEGS            R0, R1
                LDR             R7, [SP,#0x38]
                MOVS            R2, #0
                MVNS            R3, R5
                CMP             R3, R7
                MOV.W           R3, #0
                IT LS
                MOVLS           R3, #1
                CMP             R0, #0
                MOV.W           R0, #0
                ITT EQ
                MOVEQ           R0, #1
                MOVEQ           R0, R3
                CMP             R0, #0
                BNE.W           return_ERROR_MALFORMED
                ADDS            R0, R7, R5
                MOV.W           R3, #0
                ADC.W           R1, R1, #0
                CMP             R0, R7
                IT CC
                MOVCC           R3, #1
                CMP             R1, #0
                IT NE
                MOVNE           R3, R2
                CMP             R3, #0
                BNE.W           call_abort
                BLX             _Znaj   ; operator new[](uint)
                MOV             R6, R0
                CBZ             R7, loc_81F86
                LDR             R1, [SP,#0x3C]
                MOV             R0, R6
                MOV             R2, R7
                BLX             __aeabi_memcpy
```

Fixed without the patch, too!

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```

All overflows trigger an abort now, exploit is *still* prevented!

```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R7, [SP,#0x38]
LDREQ           R7, [SP,#0x38]
MOV             R8, R5
LDRD.W          R5, R1, [SP,#0xF0]
MOVS            R3, #0
MOVS            R2, #0
ADDS            R0, R7, R5
ADC.W           R1, R1, #0
CMP             R0, R7
IT CC
MOVCC           R3, #1
CMP             R1, #0
IT NE
MOVNE           R3, R2
CMP             R3, #0
BNE.W           call_abort
BLX             _Znaj   ; operator new[](uint)
MOV             R6, R0
CBZ             R7, loc_81F62
LDR             R1, [SP,#0x3C]
MOV             R0, R6
MOV             R2, R7
BLX             __aeabi_memcpy
```

UBSan applied to libstagefright

In summary:

- UBSan with original patch: no integer overflow, stops exploit!
- UBSan with *no patch*: no integer overflow, stops exploit!

SEEMS LEGIT.

What went wrong?

- The provided patch works if `chunk_size` is a `size_t`
- It is unfortunately, a `uint64_t` even on 32-bit platforms
- So, `chunk_size` can be larger than `SIZE_MAX`
- And overflow still can occur
- Exploit details here: https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html
- Getting these things right is time consuming and error prone.
  - That's why we make the compiler do it for us!

Other issues

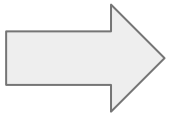
- Mediaserver was a great target for exploitation
- Many permissions due to its monolithic design
  - Bluetooth
  - Camera
  - Graphics
  - Internet
  - Microphone
  - Phone
- tl;dr exploit a codec, pwn everything

We fixed that too :)

Android M - Capabilities per process

MediaServer

Audio devices
Bluetooth
Camera Device
Custom Vendor Drivers
DRM hardware
FM Radio
GPU
IPC connection to Camera daemon
mmap executable memory
Network sockets
Read access to app-provided files
Read access to conf files
Read/Write access to media
Secure storage
Sensor Hub connection
Sound Trigger Devices



Android N - Capabilities per process

AudioServer

Audio Devices
Bluetooth
Custom vendor drivers
FM radio
Read/Write access to media

MediaCodecService

GPU

CamerServer

Camera Device
GPU
IPC connections to Camera daemon
Sensor Hub Connection

MediaServer

GPU
Network Sockets
Read access to app-provided files
Read access to conf files

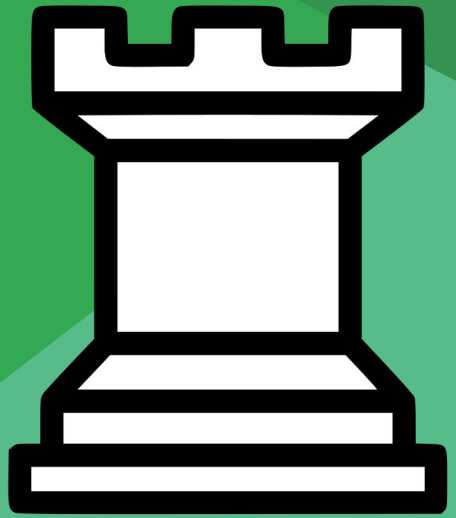
MediaDrmServer

DRM hardware
Mmap executable memory
Network sockets
Secure storage

ExtractorService

None

Reducing attack surface: IOCTL filtering



Driver code is hard

A Wi-Fi kernel driver in Android 6.0.1 before 2016-03-01 allows attackers to gain privileges via a crafted application, only needs android.permission.INTERNET and is permissible from untrusted-app (CVE-2016-0820)

- Data section overflow, caused by lack of bounds checking on copy_from_user
- Data section contains function pointers, called by Java functions
- TL;DR Memory corruption + Information leak + function pointer overwrite + ret2dir = :D

Vulnerable IOCTL call #1: IOCTL_GET_STRUCT

- *Private* Wireless Extension IOCTL read function
- Used to populate an `NDIS_TRANSPORT_STRUCT`
- We're interested in subfunction `PRIV_CMD_SW_CTRL`

It isn’t always done correctly...

In drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_SW_CTRL:

    pu4IntBuf = (PUINT_32)prIwReqData->data.pointer;

    prNdisReq = (P_NDIS_TRANSPORT_STRUCT) &aucOidBuf[0];

    if (copy_from_user(&prNdisReq->ndisOidContent[0],
                       prIwReqData->data.pointer,
                       prIwReqData->data.length)) {
        status = -EFAULT;
        break;
    }
```

This is completely user controllable (and unchecked).

It isn’t always done correctly...

In drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_SW_CTRL:

    pu4IntBuf = (PUINT_32)prIwReqData->data.pointer;

    prNdisReq = (P_NDIS_TRANSPORT_STRUCT) &aucOidBuf[0];

    if (copy_from_user(&prNdisReq->ndisOidContent[0],
                       prIwReqData->data.pointer,
                       prIwReqData->data.length)) {
        status = -EFAULT;
        break;
    }
```

Which means this can be overflowed...

It isn’t always done correctly...

In drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_SW_CTRL:

    pu4IntBuf = (PUINT_32)prIwReqData->data.pointer;

    prNdisReq = (P_NDIS_TRANSPORT_STRUCT) &aucOidBuf[0];

    if (copy_from_user(&prNdisReq->ndisOidContent[0],

                       prIwReqData->data.pointer,

                       prIwReqData->data.length)) {

        status = -EFAULT;

        break;

    }
```

Which will be an overwrite here...

...and can be taken advantage of

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

...which is in this data area...

...and can be taken advantage of

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

...and we'd like to get to this...

...and can be taken advantage of

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

...but this is all compiler dependent (and blind manipulation would likely cause a kernel panic).

Enter vulnerable IOCTL call #2: PRIV_GET_INT

- *Another Private* Wireless Extension IOCTL read function
- We’re interested in subfunction `PRIV_CMD_GET_DEBUG_CODE`
- Used to retrieve the value of `gucBufDbgCode`

Which gives us a bit more information...

drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_GET_DEBUG_CODE:
{
	wlanQueryDebugCode(prGlueInfo->prAdapter);
	kalMemSet(gucBufDbgCode, '.', sizeof(gucBufDbgCode));
	if (copy_to_user(prIwReqData->data.pointer, gucBufDbgCode, prIwReqData->data.length)) {
		return -EFAULT;
	}
	else
		return status;
}
```

This also is completely user controllable (and unchecked).

Which gives us a bit more information...

drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_GET_DEBUG_CODE:
{
	wlanQueryDebugCode(prGlueInfo->prAdapter);
	kalMemSet(gucBufDbgCode, '.', sizeof(gucBufDbgCode));
	if (copy_to_user(prIwReqData->data.pointer, gucBufDbgCode, prIwReqData->data.length)) {
		return -EFAULT;
	}
	else
		return status;
}
```

This is found in the data area we are interested in...

Which gives us a bit more information...

drivers/misc/mediatek/conn_soc/drv_wlan/mt_wifi/wlan/os/linux/gl_wext_priv.c:

```
case PRIV_CMD_GET_DEBUG_CODE:
{
	wlanQueryDebugCode(prGlueInfo->prAdapter);
	kalMemSet(gucBufDbgCode, '.', sizeof(gucBufDbgCode));
	if (copy_to_user(prIwReqData->data.pointer, gucBufDbgCode, prIwReqData->data.length)) {
		return -EFAULT;
	}
	else
		return status;
}
```

...we can effectively read the entire data area into this buffer, which is copied to userspace...

And the result...

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

We can easily find the function pointers corresponding to this

And the result...

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

And we can calculate the offset (and record the data) to craft our overflow

And the result...

```
.bss:C0C9EAA4 ; UINT_8 aucOidBuf[4096]
.bss:C0C9FAA4 ; UINT_8 gucBufDbgCode[1000]
.bss:C0C9FE8C ; UINT_8 g_ucMiracastMode
.bss:C0C9FE8D ; ALIGN 0x10
.bss:C0C9FE90 ; int num_bind_process_0
.bss:C0C9FE94 ; pid_t bind_pid_0[4]
.bss:C0C9FEA4 ; unsigned __int8 fgIsResetting
.bss:C0C9FEA5 ; ALIGN 4
.bss:C0C9FEA8 ; work_struct work_rst
.bss:C0C9FEB8 ; PARAM_SCAN_REQUEST_EXT_T rScanRequest
.bss:C0C9FEE4 ; UINT_8 wepBuf_0[48]
.bss:C0C9FF14 ; net_device *g_P2pPrDev
.bss:C0C9FF18 ; wireless_dev *gprP2pWdev
.bss:C0C9FF1C ; UINT_16 mode_0
.bss:C0C9FF1E ; ALIGN 0x10
.bss:C0C9FF20 ; int HifTxCnt
.bss:C0C9FF24 ; platform_device *HifAhbPDev
.bss:C0C9FF28 ; remove_card pfWlanRemove
.bss:C0C9FF2C ; probe_card pfWlanProbe
```

But, what do we overwrite this with?

Preparing Shellcode for the attack

1. Craft our shellcode in userspace
2. Find user page address of this buffer in /proc/<pid>/pagemap
3. Calculate the page frame number
    a. Kernel has mapped this page frame into user space, which results in an alias effectively being created in kernel space.
4. Calculate the kernel address from the start of physmap, PFN_MIN, PAGE_SIZE and the page frame number
5. **Bonus**: Keep the shellcode in one page to not worry about forcing contiguous alias pages

Putting it together...

- `IOCTL_GET_INT:PRIV_CMD_GET_DEBUG_CODE` to get the address we need
- Shellcode in userspace
- use ret2dir to get address in kernel
- `IOCTL_GET_STRUCT:PRIV_CMD_SW_CTRL` to do the function pointer overwrite (with shellcode ret2dir address)
- triggered with call to `pfWlanRemove` through `setWifiEnabled`
- All in the context of `untrusted_app` with nothing more than `INTERNET` permission

But, why?

No filtering on IOCTL calls: everyone can use everything

Even untrusted_app

Custom apps can exploit this vulnerability, completely reliably

SELinux (again) to the rescue!

SELinux IOCTL filtering

- IOCTL permissions were granted on an all-or-nothing basis
- IOCTL filtering allows for targeted whitelisting
  - By type (the magic number assigned to the driver)
  - And number (the command ID)
- Reduces potential attack surface: no longer an all-or-nothing thing

Mitigates CVE-2016-0820 completely

```
Reduce socket ioctl perms

Reduce the socket ioctl commands available to untrusted/isolated apps.
Neverallow accessing sensitive information or setting of network parameters.
Neverallow access to device private ioctls i.e. device specific
customizations as these are a common source of driver bugs.

Define common ioctl commands in ioctl_defines.
```

Stops the exploit before it can happen, as neither IOCTL call is permitted from `untrusted_app`

But, the vulnerability still exists...

User-Kernel Copy Mitigations

Bounds checking is hard

More common than you'd think!

CVE-2016-0820

- Mitigated by IOCTL filtering for untrusted_app
- Vulnerability still exists...
- Can still be used for privilege escalation!

So, what do we do…?

Hardened user-copy

- `copy_to_user()` and `copy_from_user()` are used quite often
- Bounds checking is not always performed, and security vulnerabilities tend to be the result
- `PAX_USERCOPY`: even poorly written code elsewhere in the kernel cannot truly copy more than it should.
- "hardened usercopy": code based on `PAX_USERCOPY` is included in the Android kernel (starting in 3.18)
- Enabled with the `CONFIG_HARDENED_USERCOPY` option.

Hardened user-copy: What does it check

- Tests to ensure:
  - Address range doesn't wrap past the end of memory
  - Kernel-space pointer is not null
  - It does not point to a zero-length kmalloc()
  - Address range does not overlap the kernel text segment
- If the address corresponds with slab-allocated object:
  - Check that what is being copied fits within the size of the object allocated
- If the address range is not handled by the slab allocator:
  - Check that the copy is either within a single or compound page and that it does not span independently allocated pages
- If the stack is involved:
  - Copied range must fit within the current process's stack
  - Copied range must fit within a single frame (if architecture supported)

Will it work on CVE-2016-0820?

```
    case PRIV_CMD_SW_CTRL:
        pu4IntBuf = (PUINT_32)prIwReqData->data.pointer;
        prNdisReq = (P_NDIS_TRANSPORT_STRUCT) &aucOidBuf[0];
        if (copy_from_user(&prNdisReq->ndisOidContent[0],
                           prIwReqData->data.pointer,
                           prIwReqData->data.length)) {
            status = -EFAULT;
            break;
        }
```

This structure passed into `copy_from_user`...

```
typedef struct _NDIS_TRANSPORT_STRUCT {
    UINT_32 ndisOidCmd;
    UINT_32 inNdisOidlength;
    UINT_32 outNdisOidLength;
    UINT_8 ndisOidContent[16];
} NDIS_TRANSPORT_STRUCT, *P_NDIS_TRANSPORT_STRUCT;
```

which is defined here, and has size ~38, which is much less than what the overflow requires

Prediction:

This feature will stop many unknown, previously exploitable vulnerabilities.

...but wait, there’s more!

Complicating kernel to user execution: PAN emulation

Privileged Access Never (PAN)

- Prevents the kernel from accessing user-space memory unless it has explicitly enabled access
  - TL;DR if a page is mapped in both user-space and kernel-space, then the kernel cannot access it unless PAN is explicitly allowed
- Available on ARM v8.1
- Emulation added to Linux Kernel for ARMv8

Emulation (on ARM v8)

- Emulates PAN by disabling translation table (specifically `TTBR0_EL1`) accesses on arm64.
- Done by pointing `TTBR0_EL1` to a zeroed, reserved Address Space ID (ASID): `0xffff000000000000`
- Access explicitly granted by temporarily setting `TTBR0_EL1` to the actual value

What it can and can’t do

- Can Stop:
  - Arbitrary access to userland (ret2usr)

- Can’t Stop:
  - Exploit stays in the kernel
  - Exploit finds a gadget that flips the PAN bit and then goes back to userland

Mitigations: Where are they?

What we are mitigating

- SELinux: Everywhere! Partial enforcing in 4.4.4, fully enforcing in 5.0+
- Integer Sanitization: Non-performance critical codecs and majority of mediaserver/ stagefright (Android 6.0+)
  - Other places as well, such as netd, minikin, keystore, init, verity, ril
- IOCTL filtering: Support added in Android 6.0, CTS enforcement starting in Android 7.0
- User copy protection
  - Android kernel 3.18, 4.1, 4.4, added in Sept 2016.
- PAN emulation
  - Android kernel 3.18, 4.1, 4.4, 4.9 added late 2016/ enabled by default (selectively) early 2017

In Conclusion...

- Android Security is watching current exploit trends
- Integrating mitigations to address exploit classes
- Making Android safer for end users...
- ...and a more challenging target for attackers

It’s your move...