Rooting every Android

From extension to exploitation

Di Shen (@returnsme), James Fang (@id3lr)
KEEN LAB TENCENT (@KEEN_LAB)

From extension to exploitation

Rooting every Android

Di Shen (@returnsme), James Fang(@idl3r)
Keen Lab Tencent (@keen_lab)

Overview

In this section we are going to discuss the implementation of Wireless Extension (WEXT) subsystem in Android. We are going to analyse the attack surface of WEXT API and reveal its potential weakness.

Wi-Fi chipsets for Android

All existing Android builds use Linux as its kernel. As a mobile-oriented system, it heavily relies on Wi-Fi and cellular data for communication. For its Wi-Fi part, Android adopted Wireless Extension (WEXT) as its standard API to manage Wi-Fi hardware.

The Wireless Extension (WEXT) was designed by Jean Tourrilhes in 1997. It has been widely used for 20 years, yet its design still meets the demanding of modern Wi-Fi hardware. It was designed a wireless API which would allow the user to manipulate any wireless networking device in a standard and uniform way.

Although a newer standard, cfg80211, is likely to replace WEXT in the near future, yet it still provides backward compatibility, which means the same user interface still exists in the kernel. It worth mentioning that newer standard doesn't necessarily mean more secured kernel, as common security flaws are still discovered in drivers adopting cfg80211 standard. In fact, WEXT drivers may arguably be more secure since it attracts more attention from researchers.

WEXT Attack Surface Analysis

The user-space interface of WEXT consists of two main components: a procfs node and ioctl through sockets.

The procfs node is /proc/net/wireless. It is read-only and supplies only statistical information on each of the wireless interfaces. From its implementation in /net/wireless/wext-proc.c, we can see only a read handler registered:

```
static const struct file_operations wireless_seq_fops = {
        .owner   = THIS_MODULE,
        .open    = seq_open_wireless,
        .read    = seq_read,
        .llseek  = seq_lseek,
        .release = seq_release_net,
};
```

From security point of view, the possibility of critical security flaw in this file is very low. This node is almost “harmless”.

On the other side, ioctl is intended to provide all the “extension” features to user-space programs. It includes setting/getting parameters on the Wi-Fi device and issuing commands to it. Following range of ioctl commands are assigned to WEXT related opeartions:

```
/* The first and the last (range) */
#define SIOCIWFIRST     0x8B00
#define SIOCIWLAST      SIOCIWLASTPRIV          /* 0x8BFF */
```

These WEXT ioctls are not magic. Each driver needs to implement them. From sys_ioctl, it takes a long call stack to reach the actual handler function in driver.

When a WEXT ioctl is called upon a socket, sys_ioctl will check the operation against its corresponding LSM hook function. In Android’s case it would be SELinux policies. If the access is allowed (or permissive), it will going to do_vfs_ioctl and vfs_ioctl. As usual, vfs_ioctl will refer to the operations table of the object, which is socket_file_ops in this case. Then the control flow will be guided to sock_ioctl.

At the begin of sock_ioctl, there is a piece of code handling ioctl commands in the range of `[SIOCIWFIRST, SIOCIWLAST]`:

```
static long sock_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
        ...
        struct net *net;
        ...
        net = sock_net(sk);
        ...
#ifdef CONFIG_WEXT_CORE
        if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST) {
                err = dev_ioctl(net, cmd, argp);
        } else
#endif
```

Note that dev_ioctl is not dedicate for WEXT ioctl commands. It also handles a wide range of other commands for socket objects. It does, however, copy the user supplied arg buffer into an ifreq structure (ifr) on stack, with a fixed length. Then, at the bottom of the function, it will hand-over the ifr object to wext_handle_ioctl:

```
int dev_ioctl(struct net *net, unsigned int cmd, void __user *arg)
```

```
{
        struct ifreq ifr;
        ...
        if (copy_from_user(&ifr, arg, sizeof(struct ifreq)))
                return -EFAULT;

        ifr.ifr_name[IFNAMSIZ-1] = 0;
        ...
        switch (cmd) {
        ...
        /*
         *      Unknown or private ioctl.
         */
        default:
                ...
                /* Take care of Wireless Extensions */
                if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST)
                        return wext_handle_ioctl(net, &ifr, cmd, arg);
                return -ENOTTY;
        }
}
```

From this point, the control flow enters the WEXT specific path. Function wext_handle_ioctl will then call wext_ioctl_dispatch, which the latter will perform an important check on permissions:

```
static int wext_permission_check(unsigned int cmd)
{
        if ((IW_IS_SET(cmd) || cmd == SIOCGIWENCODE ||
             cmd == SIOCGIWENCODEEXT) &&
            !capable(CAP_NET_ADMIN))
                return -EPERM;

        return 0;
}

...

static int wext_ioctl_dispatch(struct net *net, struct ifreq *ifr,
                               unsigned int cmd, struct iw_request_info *info,
                               wext_ioctl_func standard,
                               wext_ioctl_func private)
{
        int ret = wext_permission_check(cmd);

        if (ret)
                return ret;

        dev_load(net, ifr->ifr_name);
        rtnl_lock();
        ret = wireless_process_ioctl(net, ifr, cmd, info, standard, private);
        rtnl_unlock();

        return ret;
}
```

And IW_IS_SET is a simple macro to detect if a number is even:

```
#define IW_IS_SET(cmd)  (!((cmd) & 0x1))
```

This means all WEXT ioctl commands which are even means a set (write) operation, and requires CAP_NET_ADMIN to be performed. On other words, if a set operation is wrongfully assigned with an odd ioctl command number, the requirement of CAP_NET_ADMIN will be missing and any unprivileged user can call it providing SELinux policy allows the syscall itself. This is actually a very real problem which we will discuss in the case studies.

If the permission check is passed, wext_ioctl_dispatch will eventually call wireless_process_ioctl, which lookup a handler table of the driver against the index (`cmd - SIOCIWFIRST`) and call the corresponding handler function.

In conclusion, WEXT ioctl commands are widely available to unprivileged users if SELinux (or other LSM modules) allows the access. It employs an "odd" rule to identify privileged and unprivileged commands, which could be wrongfully implemented and cause unprivileged access.

Use device specific vulnerabilities to root them all

In the previous section, we discussed why WEXT ioctl commands are good candidate to attack the kernel. In the next sections we will present three vulnerabilities affecting Qualcomm, Broadcom and Mediatek Wi-Fi deviecs respectively, which covers majority of Android devices.

Case Studies

Stack overflow vulnerability in Qualcomm WEXT

CVE-2015-0570 is a stack overflow vulnerability reported by anonymous researcher last year. Before digging into the bug itself, we need to understand how this bug survives the Stack Smashing Protection (SSP) of gcc.

SSP is recommended to be enabled when compiling kernel. In theory, any function has local array can be potential victim of stack overflow vulnerability. However, to reduce the overhead of stack canary check, several rules are made to restrict the number of functions which SSP will be applied. One rule is that, if all local arrays of a function are less than 8 bytes long, it won't be applied. Gcc is capable of generate a warning when a function is not protected by SSP, which is off by default, since obviously it will spam the warning output.

By enabling the warning manually (`-Wstack-protector`), we can read the following output:

```
/msm/bullhead-src/drivers/staging/qcacld-2.0/CORE/HDD/src/wlan_hdd_wext.c:
In function 'wlan_hdd_set_filter':

/msm/bullhead-src/drivers/staging/qcacld-
2.0/CORE/HDD/src/wlan_hdd_wext.c:8315:5: warning: stack protector not
protecting function: all local arrays are less than 8 bytes long [-Wstack-
protector]
```

The function body declared two local members on stack:

```
int wlan_hdd_set_filter(hdd_context_t *pHddCtx, tpPacketFilterCfg pRequest,
                            tANI_U8 sessionId)
{
    tSirRcvPktFilterCfgType    packetFilterSetReq = {0};
    tSirRcvFltPktClearParam    packetFilterClrReq = {0};
    ...
}
```

And the definitions of these two structures can be found in sirApi.h:

```
#define    SIR_MAX_FILTER_TEST_DATA_LEN       8
...
typedef struct sSirRcvPktFilterFieldParams
{
  eSirRcvPktFltProtocolType        protocolLayer;
  eSirRcvPktFltCmpFlagType         cmpFlag;
  /* Length of the data to compare */
  tANI_U16                         dataLength;
  /* from start of the respective frame header */
  tANI_U8                          dataOffset;
  /* Reserved field */
```

```
  tANI_U8                          reserved;
  /* Data to compare */
  tANI_U8                          compareData[SIR_MAX_FILTER_TEST_DATA_LEN];
  /* Mask to be applied on the received packet data before compare */
  tANI_U8                          dataMask[SIR_MAX_FILTER_TEST_DATA_LEN];
}tSirRcvPktFilterFieldParams, *tpSirRcvPktFilterFieldParams;

typedef struct sSirRcvPktFilterCfg
{
  tANI_U8                         filterId;
  eSirReceivePacketFilterType     filterType;
  tANI_U32                        numFieldParams;
  tANI_U32                        coalesceTime;
  tSirMacAddr                     selfMacAddr;
  tSirMacAddr                     bssId; //Bssid of the connected AP
  tSirRcvPktFilterFieldParams     paramsData[SIR_MAX_NUM_TESTS_PER_FILTER];
}tSirRcvPktFilterCfgType, *tpSirRcvPktFilterCfgType;
```

It is noticed that there are two arrays declared inside tSirRcvPktFilterFieldParams, which are encapsulated in tSirRcvPktFilterCfgType. Both array has 8 bytes. It seems that the gcc's decision was wrong. Further investigation shows that gcc was not able to handle nested structure array properly. This might be a bug in SSP implementation which worth further investigation.

The relatively low efficiency of SSP has been and google posted a blog in June 2016: http://android-developers.blogspot.com/2016/07/protecting-android-with-more-linux.html. In this blog, Google recommended to use Strong Stack Smashing Protection (-fstack-proetector-strong) instead of the regular one. This feature is available since gcc 4.9.

Back to the vulnerability itself, it is quite obvious. The data copy loop in function wlan_hdd_set_filter does not check data length:

```
int wlan_hdd_set_filter(hdd_context_t *pHddCtx, tpPacketFilterCfg pRequest, tANI_U8
sessionId)
{
    tSirRcvPktFilterCfgType    packetFilterSetReq = {0};
    ...

    switch (pRequest->filterAction)
    {
        case HDD_RCV_FILTER_SET:
            ...
            for (i=0; i < pRequest->numParams; i++)
            {
                ...
                packetFilterSetReq.paramsData[i].dataLength =
pRequest->paramsData[i].dataLength;
                ...
                memcpy(&packetFilterSetReq.paramsData[i].compareData,
                        pRequest->paramsData[i].compareData,
pRequest->paramsData[i].dataLength);
                memcpy(&packetFilterSetReq.paramsData[i].dataMask,
```

```
                        pRequest->paramsData[i].dataMask,
pRequest->paramsData[i].dataLength);
                ...
            }
        ...
    }
    return 0;
}
```

According to the call stack, pRequest is exactly the arg supplied by user when calling ioctl. By crafting a malicious pRequest, which is fully controllable by the user, we can overflow the stack of wlan_hdd_set_filter and gain control of LR.

Another remaining issue is that, to defeat PXN, which is widely enabled on arm64 devices, we need to construct a JOP chain. For this case, we are controlling X29 and X19 registers, which is not ideal for JOP. Luckily, we were able to find the following pivot gadget:

```
bin_page_mkwrite:
        A1 1F 40 F9                 LDR             X1, [X29,#0x38]
        E0 03 14 AA                 MOV             X0, X20
        60 02 3F D6                 BLR             X19
```

After controlling X0 and X1, we can easily build 2 JOP chains to leak kernel SP and overwrite addr_limit. Our choice of gadgets are:

Leak SP:

```
shm_sync:
        05 08 40 F9                 LDR             X5, [X0,#0x10]
        A0 14 40 F9                 LDR             X0, [X5,#0x28]
        04 38 40 F9                 LDR             X4, [X0,#0x70/0x78]
        A0 02 80 12                 MOV             W0, #0xFFFFFFEA
        64 00 00 B4                 CBZ             X4, loc_FFFFFFC0003DFB10
        E0 03 05 AA                 MOV             X0, X5
        80 00 3F D6                 BLR             X4
snd_pcm_common_ioctl1:
        03 08 40 F9                 LDR             X3, [X0,#0x10]
        E0 03 1C AA                 MOV             X0, X28
        60 00 3F D6                 BLR             X3
__spi_async:
        20 08 00 F9                 STR             X0, [X1,#0x10]
        22 34 00 B9                 STR             W2, [X1,#0x34]
        A2 78 41 F9                 LDR             X2, [X5,#0x2F0/0x380]
        40 00 3F D6                 BLR             X2
```

Overwrite addr_limit:

```
shm_sync:
        05 08 40 F9                 LDR             X5, [X0,#0x10]
        A0 14 40 F9                 LDR             X0, [X5,#0x28]
        04 38 40 F9                 LDR             X4, [X0,#0x70/0x78]
        A0 02 80 12                 MOV             W0, #0xFFFFFFEA
        64 00 00 B4                 CBZ             X4, loc_FFFFFFC0003DFB10
        E0 03 05 AA                 MOV             X0, X5
        80 00 3F D6                 BLR             X4

df_bcc_func:
```

```
       03 04 40 F9                LDR             X3, [X0,#8]
       00 18 40 F9                LDR             X0, [X0,#0x30]
       60 00 3F D6                BLR             X3
__spi_async:
       20 08 00 F9                STR             X0, [X1,#0x10]
       22 34 00 B9                STR             W2, [X1,#0x34]
       A2 78 41 F9                LDR             X2, [X5,#0x2F0/2F8/380]
       40 00 3F D6                BLR             X2
```

Data section overflow vulnerability in MTK WEXT

The discovery of CVE-2015-0570 put our attention focus on WEXT attack surface. Soon after that we found another vulnerability in MediaTek WEXT. That bug was obviously exploitable, so I finish developing the exploit in two days, and no hardcoded kernel symbol is needed in its exploit. At that time, we didn't report it to Google yet. Taking into account the bug can be easily discovered by code auditing, I'm not surprised when I noticed that another researcher Mark Brand of Google project zero already reported it to Google in Dec. 2015.

The CVE number of 2nd case is CVE-2016-0820. It affected all mediaek-based devices.

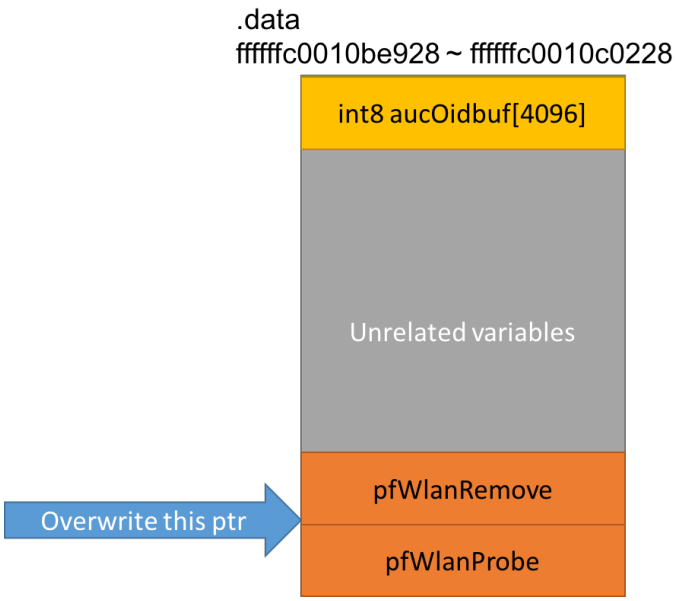
The overflow

When one of WEXT handler ***priv_get_struct*** process PRINV_CMD_SW_CTRL command, there is no boundary protection of the copy length when it call ***copy_from_user.*** The destination of memory copy is ***aucOidbuf*** which has 4096 bytes in data section. The length of copied buffer is ***prIwReqData->data.length*** which is provided by user and can be any value.

```
1437        case PRIV_CMD_SW_CTRL:
1438            pu4IntBuf = (PUINT_32)prIwReqData->data.pointer;
1439            prNdisReq = (P_NDIS_TRANSPORT_STRUCT) &aucOidBuf[0];
1440            //kalMemCopy(&prNdisReq->ndisOidContent[0],
        prIwReqData->data.pointer, 8);
1441            if (copy_from_user(&prNdisReq->ndisOidContent[0],
1442                               prIwReqData->data.pointer,
1443                               prIwReqData->data.length)) {
1444                status = -EFAULT;
1445                break;
1446            }
```

How to exploit

A classical idea to achieve kernel code execution is overwriting a global function pointer located behind ***aucOidbuf*** and then trigger some one to call it. In this case I'll try to overwrite ***pfWlanRemove.*** To avoid a kernel crash, I'd better not corrupt other unrelated global variables between ***aucOidbuf*** and ***pfWlanRemove***. So I need to know the offset of ***pfWlanRemove,*** and the value of these unrelated variables. To meet these requirements, leaking the value behind ***aucOidbuf*** is necessary.



Leaking values

Fortunately, another command PRIV_CMD_GET_DEBUG_CODE completed the task perfectly. There is no boundary check when call ***copy_to_user.*** So we can get the value of variables behind ***gucBufDbgCode*** which is the variable just behind ***aucOidBuf***.

```
1097    case PRIV_CMD_GET_DEBUG_CODE:
1098        {
1099            wlanQueryDebugCode(prGlueInfo->prAdapter);
1100                kalMemSet(gucBufDbgCode, '.', sizeof(gucBufDbgCode));
1101                if (copy_to_user(prIwReqData->data.pointer, gucBufDbgCode,
    prIwReqData->data.length)) {
1102                    return -EFAULT;
1103                }
1104                else
1105                    return status;
1106        }
```

Kernel code execution

Now it's time to allocate some pages in user mode and copy shellcode to these pages. Then get the direct mapped address of these pages in kernel (a.k.a. ret2dir). Note that these pages are EXECUTABLE in kernel space of MTK devices.

Then overwrite ***pfWlanRemove*** with kernel address of shellcode. And finally call Java API ***wifi.setWifiEnabled(false)*** so that system process "mtk_wmtd" may call ***pfWlanRemove*** to execute shellcode in kernel space.

Use-After-Free vulnerability in Broadcom WEXT

Case study 3 is a Use-after-free due to race condition in Broadcom WEXT, both the bug and its exploitation is much complicated that previous two case. There are two separated issue involved in this case. The first one is CVE-2016-2475, which is a lack of privileged check while processing WEXT ioctl command for Android. The second one is the Use-after-free when call ***wl_android_wifi_off*** concurrently. These issues affected all premium-end Android phones like Samsung Galaxy series, Huawei Mate series, Google Nexus 6p, etc.

An interesting fact is that the second issue is discovered by running test code while pressing Wi-Fi button on and off repeatedly and crazily. And then kernel crashed and the crash is reproducible. After analyzing the crash, I realized I found a UAF bug accidently.

CVE-2016-2475

This issue exposes a surface for attacker. In ***dhd_ioctl_entry, wl_android_priv_cmd*** can be called with insufficient privileges. The command *SIOCDEVPRIVATE+1* will be processed at position 1, it's too late to check the privilege of process at position 2.

```
1  static int dhd_ioctl_entry(struct net_device *net, struct ifreq *ifr, int cmd)
2  {
3
4      ...snip...
5
6      if (cmd == SIOCDEVPRIVATE+1) {   //position 1
7          ret = wl_android_priv_cmd(net, ifr, cmd);
8          dhd_check_hang(net, &dhd->pub, ret);
9          DHD_OS_WAKE_UNLOCK(&dhd->pub);
10         return ret;
11     }
12
13     if (cmd != SIOCDEVPRIVATE) {
14         DHD_PERIM_UNLOCK(&dhd->pub);
15         DHD_OS_WAKE_UNLOCK(&dhd->pub);
16         return -EOPNOTSUPP;
17     }
18
19     ...snip...
20
21     if (!capable(CAP_NET_ADMIN)) {     //position 2
22         bcmerror = BCME_EPERM;
23         goto done;
24     }
25
26     ...snip...
27
28     bcmerror = dhd_ioctl_process(&dhd->pub, ifidx, &ioc, local_buf);
```

A large number of commands for Android will be processed in ***wl_android_priv_cmd*** , the most attractive to me are CMD_START/CMD_STOP which can be used to enable/disable Wi-Fi directly. That UAF bug is introduced by the implementation of CMD_STOP.

```
 1  int wl_android_priv_cmd(struct net_device *net, struct ifreq *ifr, int cmd){
 2      ...snip...
 3      if (strnicmp(command, CMD_START, strlen(CMD_START)) == 0) {
 4          bytes_written = wl_android_wifi_on(net);
 5      }
 9      if (!g_wifi_on) {
10          ret = 0;
11          goto exit;
12      }
13      if (strnicmp(command, CMD_STOP, strlen(CMD_STOP)) == 0) {
14          bytes_written = wl_android_wifi_off(net, FALSE);
15      }
25      ...snip...
26      return ret;
27  }
```

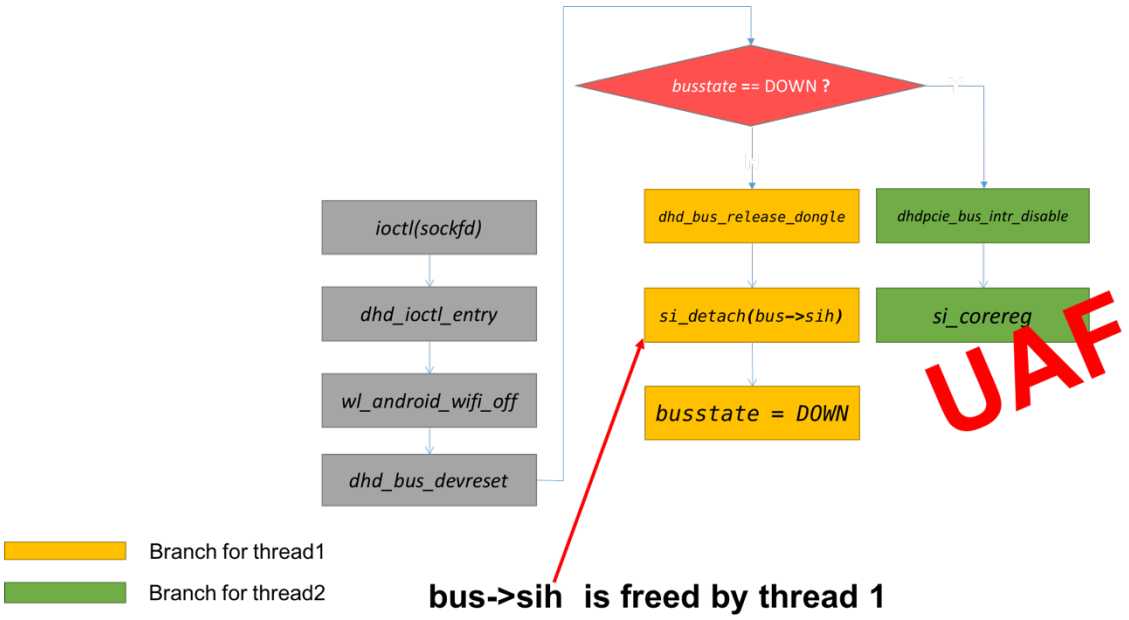
Android-ID-24739315

Here's come the second issue, it has not a CVE number and never appeared in Android Security Bulletin, but it's absolutely exploitable. The patch is quite simple. When ***dhd_bus_devreset*** is called and the state of Wi-Fi bus has been down, do not call ***dhdpcie_bus_intr_disable*** any more.

```
net: wireless: bcmdhd: remove unnecessary PCIe memory access when BUS down.

In case PCIe BUS already down, we're not supposed to do access BAR0
area in any reason. One instance seen on test that made kernel panic.
removed disable irq calling which is useless in bus down case.

bug=24739315

Change-Id: I474e08c14c4dec0f4cc4cd207f29fef32e85ead7
Signed-off-by: Insun Song <isong@broadcom.com>
```

```
diff --git a/drivers/net/wireless/bcmdhd/dhd_pcie.c b/drivers/net/wireless/bcmdhd/dhd_pcie.c
index 21bbe54..1adffc3 100644
--- a/drivers/net/wireless/bcmdhd/dhd_pcie.c
+++ b/drivers/net/wireless/bcmdhd/dhd_pcie.c

@@ -2472,7 +2472,6 @@
                        bus->dhd->busstate = DHD_BUS_DOWN;
                } else {
                        if (bus->intr) {
-                               dhdpcie_bus_intr_disable(bus);
                                dhdpcie_free_irq(bus);
                        }
```

If two threads call ***wl_android_wifi_off*** simultaneously, the first thread will go yellow branch, may free the ***si_info*** and set bus state to DOWN, the second thread will go green branch later, reference freed ***struct si_info.***

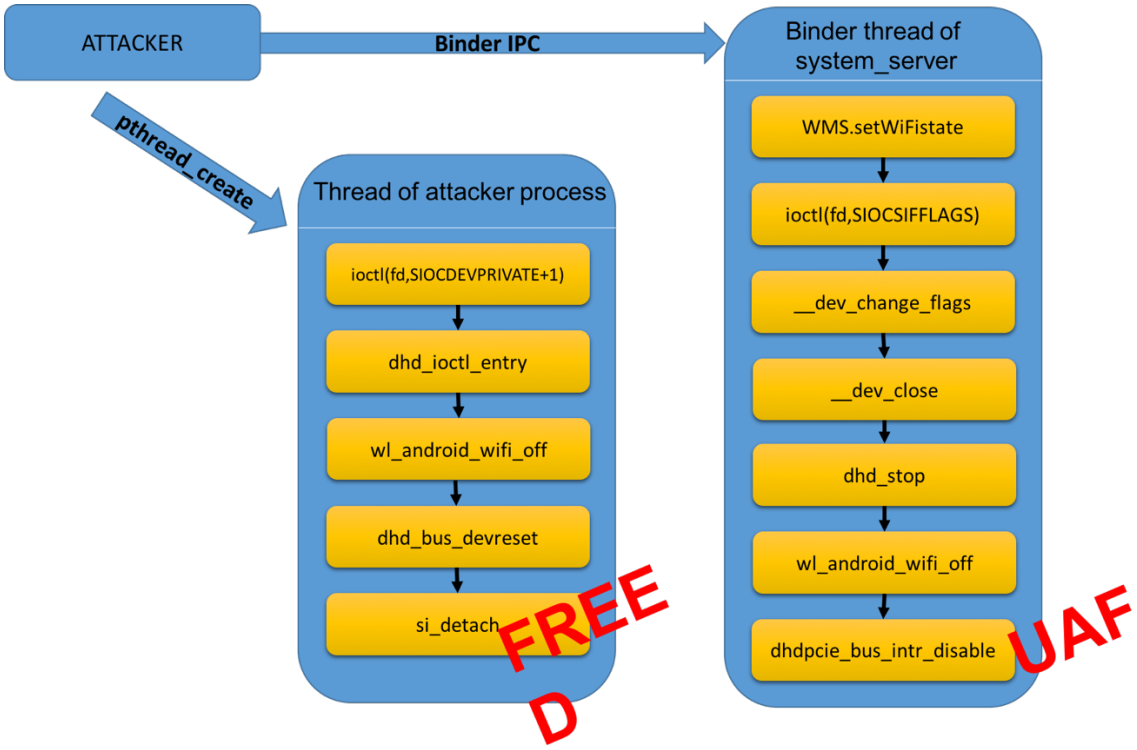


How to trigger

The prerequisites of triggering the UAF bug is invoking ***wl_android_wifi_off*** simultaneously, however, attacker can not invoke ***wl_android_wifi_off*** via ***dhd_ioctl_entry*** concurrently because it's locked by DHD_PERIM_LOCK. I have to find another way.

```
1  static int
2  dhd_ioctl_entry(struct net_device *net, struct ifreq *ifr, int cmd)
3  {
4      dhd_info_t *dhd = DHD_DEV_INFO(net);
5      dhd_ioctl_t ioc;
6      int bcmerror = 0;
7      int ifidx;
8      int ret;
9      void *local_buf = NULL;
10     u16 buflen = 0;
11
12     DHD_OS_WAKE_LOCK(&dhd->pub);
13     DHD_PERIM_LOCK(&dhd->pub);
14 ..snip..
15     if (cmd == SIOCDEVPRIVATE+1) {
16         ret = wl_android_priv_cmd(net, ifr, cmd);
17         dhd_check_hang(net, &dhd->pub, ret);
18         DHD_OS_WAKE_UNLOCK(&dhd->pub);
19         return ret;
20     }
21 ..snip..
22 done:
23     if (local_buf)
24         MFREE(dhd->pub.osh, local_buf, buflen+1);
25
26     DHD_PERIM_UNLOCK(&dhd->pub);
27     DHD_OS_WAKE_UNLOCK(&dhd->pub);
28 }
```

Fortunately, ***wl_android_wifi_off*** also can be called via ***dhd_stop***, the code flow is ***devnet_ioctl(sockfd,SIOCSIFFLAGS) -> __dev_change_flags -> __dev_close -> dhd_stop -> wl_android_wifi_off*** . Even though calling ***ioctl(sockfd,SIOCSIFFLAGS)*** requires CAP_NET_ADMIN privilege, any local application is able to ask system_server to call a Java method ***setWiFistate*** via binder IPC. System_server will handle request in a binder thread, and finally call ***wl_android_wifi_off.***



To trigger the UAF, the binder thread has to enter ***wl_android_wifi_off*** soon after attacker's thread enter it. The window is small. If the re-filled object meet following requirements, it will reach a macro INTR_OFF, ***intrsoff_fn*** is a function pointer can be controlled by attacker.

```
void
dhdpcie_bus_intr_disable(dhd_bus_t *bus)
    if (bus) {

        if ((bus->sih->buscorerev == 2) || (bus->sih->buscorerev == 6) ||
            (bus->sih->buscorerev == 4)) {
            dhpcie_bus_mask_interrupt(bus);
        }
        else if (bus->sih) {
            si_corereg(bus->sih, bus->sih->buscoreidx, PCIMailBoxMask,
                bus->def_intmask, 0);
        }
    }
}
```

buscorerev != 2,4,6

```
uint
si_corereg(si_t *sih, uint coreidx, uint regoff, uint mask, uint
val)
{
    if (CHIPTYPE(sih->socitype) == SOCI_SB)
        return sb_corereg(sih, coreidx, regoff, mask, val);
    else if ((CHIPTYPE(sih->socitype) == SOCI_AI) ||
  (CHIPTYPE(sih->socitype) == SOCI_NAI))
        return ai_corereg(sih, coreidx, regoff, mask, val);
    else if (CHIPTYPE(sih->socitype) == SOCI_UBUS)
        return ub_corereg(sih, coreidx, regoff, mask, val);
    else {
        ASSERT(0);
        return 0;
    }
}
```

socitype == SOCI_SB

- bus->sih is the freed si_info

```
uint sb_corereg(si_t *sih, uint coreidx, uint regoff, uint mask,
uint val)
{
    ...snip...

    if (BUSTYPE(sii->pub.bustype) == SI_BUS) {
        fast = TRUE;
        ...snip...
    } else if (BUSTYPE(sii->pub.bustype) == PCI_BUS) {
        fast = TRUE;
        ...snip...
    }

    if (!fast) {
        INTR_OFF(sii, intr_val);
```

bustype != SI_BUS,PCI_BUS

```
#define INTR_OFF(si, intr_val) \
    if ((si)->intrsoff_fn && (cores_info)->coreid[(si)->curidx]\ ==
(si)->dev_coreid) {    \
        intr_val = (*(si)->intrsoff_fn)((si)->intr_arg); }
```

- si_info->intrsoff_fn is the crafted function pointer I can control

It's a UAF due to race condition and the window is very small. In the following log you can know that the object was free at 872.481513s and reused at 872.592760s. Attacker only has 0.02s to re-fill the object and execute code in kernel. This is a big challenge.

```
<4>[  872.481513] si_detach free si ffffffc058ae1100;cores_info ffffffc0ac815
<4>[  872.481526] dhdpcie_bus_release_dongle Exit
<4>[  872.481574] dhdpcie_stop_host_pcieclock Enter:
<6>[  872.496619] msm_pcie_disable: PCIe: Assert the reset of endpoint of RC1
<4>[  872.501069] dhdpcie_stop_host_pcieclock Exit:
<4>[  872.501081] dhd_bus_devreset:  WLAN OFF Done
<4>[  872.501094] wifi_platform_set_power = 0
<6>[  872.501104] dhd_wlan_power Enter: power off
<4>[  872.501383] __dev_change_flags  dev: wlan0 flags 1042
<4>[  872.501598] dhd_deferred_work_handler: event to handle 24
<4>[  872.501615] dhd_set_mcast_list_handler: interface info not available/dc
<4>[  872.501626] dhd_deferred_work_handler: event to handle 0
<4>[  872.501634] dhd_deferred_work_handler: No event to handle 0
<4>[  872.502660] dhd_stop: Enter ffffffc0b11d5000
<4>[  872.502672] wl_android_wifi_off in
<4>[  872.502684] dhd_prot_ioctl : bus is down. we have nothing to do
<4>[  872.502695] dhd_net_bus_devreset: wl down failed
<4>[  872.502704] dhd_bus_devreset: == Power OFF ==
<4>[  872.502715] dhdpcie_bus_intr_disable Enter
<4>[  872.502760] sb_corereg sii ffffffc058ae1100,cores_info ffffffc0ac815000
```

Racing and object re-filling

If racing failed, nothing will happen except an error returned. If racing succeeded but re-filling failed, kernel may crash. So we have to find a way to spray kernel heap efficiently and quickly. And also, in this case we'd better fully control the content and length of the re-filled objects. I used to use sendmmsg to spray kernel heap, and this time it's still working.

Heap spraying by sendmmsg

Firstly, create two processes as server and client, and send bytes over a TCP connection using sendmmsg. Let msghdr->msg_control point to the content you want to spray in kernel. As server never respond the sendmsg request from client, the kernel buffer of msg_control will permanently stay in kmalloc heap.

```
struct msghdr {
    void    *    msg_name;    /* Socket name            */
    int         msg_namelen;    /* Length of name        */
    struct iovec *    msg_iov;    /* Data blocks            */
    __kernel_size_t    msg_iovlen;    /* Number of blocks        */
    void     *    msg_control; /* Per protocol magic (eg BSD file descriptor passing) */
    __kernel_size_t    msg_controllen;    /* Length of cmsg list */
    unsigned int    msg_flags;
};
```

It has a 90% success rate to re-fill the freed object in 0.02s. The data and length of sprayed object can be fully controlled. Fortunately, the freed ***si_info*** is allocated in kmalloc-256. This approach will be not working if the object is located in kmalloc-512, since sendmmsg will allocate other 512-sized objects as an interference with spraying.

Happy rooting

Now we have achieved kernel code execution, gaining root could be quite easy, just build JOP gadgets to manipulate credential of attacker's process, disable SELinux and bypass some vendor specific mitigation (e.g. Samsung KNOX's Real-time Kernel Protection).

Google's latest mitigation

The discovery of CVE-2015-0570 put researchers' attention on WEXT, while CVE-2016-0820 drove Google to reduce socket ioctl permissions further. Google's patch for SELinux policy has a same bug id with CVE-2016-0820

```
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.

Bug: 26267358
Change-Id: Ic5c0af066e26d4cb2867568f53a3e65c5e3b5a5d
```

diff --git a/Android.mk b/Android.mk

Now only a limited set of socket ioctls defined in ***upriv_sock_ioctls*** can be accessed by unprivileged apps.

```
# only allow unprivileged socket ioctl commands
allow untrusted_app self:{ rawip_socket tcp_socket udp_socket } unpriv_sock_ioctls;

# Allow GMS core to access perfprofd output, which is stored
# in /data/misc/perfprofd/. GMS core will need to list all
```

I'm not surprised that private WEXT ioctls are removed from ***upriv_sock_ioctls.***

```
 # socket ioctls allowed to unprivileged apps
 define(`unpriv_sock_ioctls', `
 {
-# all socket ioctls except the Mac address SIOCGIFHWADDR 0x8927
-0x8900-0x8926 0x8928-0x89ff
-# all wireless extensions ioctls except get/set essid
-# IOCSIWESSID 0x8B1A SIOCGIWESSID 0x8B1B
-0x8B00-0x8B19 0x8B1C-0x8BFF
+# Socket ioctls for gathering information about the interface
+SIOCGIFNAME SIOCGIFCONF SIOCGIFFLAGS SIOCGIFADDR SIOCGIFBRDADDR
+SIOCGIFNETMASK SIOCGIFMTU SIOCGIFCOUNT SIOCGIFTXQLEN
+# Wireless extension ioctls. Primarily get functions.
+SIOCGIWNAME SIOCGIWFREQ SIOCGIWMODE SIOCGIWSENS SIOCGIWRANGE SIOCGIWPRIV
+SIOCGIWSTATS SIOCGIWSPY SIOCSIWTHRSPY SIOCGIWTHRSPY SIOCGIWRATE SIOCGIWRTS
+SIOCGIWFRAG SIOCGIWTXPOW SIOCGIWRETRY SIOCGIWPOWER
 # commonly used TTY ioctls
-0x5411 0x5451
+TIOCOUTQ FIOCLEX
 }')
```

So if you want to call private WEXT ioctl on an updated Nexus device, I'm afraid you may get following error message from kernel:

avc: denied { ioctl } for pid=8567 comm="poc" path="socket:[156925]" dev="sockfs" ino=156925 ***ioctlcmd=89f1*** *scontext=u:r:shell:s0 tcontext=u:r:shell:s0 tclass=tcp_socket permissive=0*

Conclusion

After we used three exploits mentioned above to root a large number of devices with Qualcomm, Mediatek, and Broadcom Wi-Fi chipsets, I believe WEXT private ioctl was once an awesome attack surface on Android kernel. Code in Linux kernel is safe but vendor code is still buggy. And Google really did a good job on surface reduction in 2016. "Protecting Android with more Linux kernel defenses" shows that what Google's Android security team have done this year.

In summary, rooting Android is becoming more and more challenging. To root Android devices again in future, mining other attack surfaces little known like WEXT private ioctl could be helpful. Discovering universal vulnerability in generic syscall like CVE-2015-1805 is another option. Compromising a privileged process first and attacking devices which are only accessible to privileged process is also reasonable. But compromising a privileged process in user space is just another hard work.