Exploiting Qualcomm WLAN And Modem Over-The-Air

Xiling Gong, Peter Pi
Tencent Blade Team
About Us

Xiling Gong (@GXiling)
Senior security researcher at Tencent Blade Team.
Focus on Android Security, Qualcomm Firmware Security.
Speaker of BlackHat, CanSecWest.

Peter Pi(@tencent_blade)
Senior security researcher at Tencent Blade Team.
Find many vulnerabilities of vendors like Google, Microsoft, Apple, Qualcomm, Adobe and Tesla.
The #1 Researcher of Google Android VRP in year 2016.
Speaker of BlackHat, CanSecWest, HITB, GSEC and Hitcon.
About Tencent Blade Team

• Founded by Tencent Security Platform Department in 2017

• Focus on security research in the areas of AIoT, Mobile devices, Cloud virtualization, Blockchain, etc

• Report 200+ vulnerabilities to vendors such as Google, Apple, Microsoft, Amazon

• We talked about how to break Amazon Echo at DEFCON26

• Blog: https://blade.tencent.com
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• Stability of Exploitation
• Conclusions
Agenda

• Introduction and Related Work
  • The Debugger
  • Reverse Engineering and Attack Surface
  • Vulnerability and Exploitation
  • Escaping into Modem
  • Escaping into Kernel
  • Stability of Exploitation
  • Conclusions
Introduction

• Broadcom WIFI Chip
  • 2017, Gal Beniamini
    • Over The Air: Exploiting Broadcom’s Wi-Fi Stack
  • 2017, Nitay Artenstein, BlackHat USA 2017
    • BROADPWN: REMOTELY COMPROMISING ANDROID AND IOS VIA A BUG IN BROADCOM’S WI-FI CHIPSETS

• Marvel WIFI Chip
  • 2019, Denis Selyanin
    • Zero Nights 2018, Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE
    • Blog 2019, Remotely compromise devices by using bugs in Marvell Avastar Wi-Fi: from zero knowledge to zero-click RCE

• How about Qualcomm WIFI?
Qualcomm WLAN (MSM8998)

- Application
- Android Framework
- Wifi Demon

- Full MAC Layer
- QCACLD2/3

- Linux Kernel

- Baseband Subsystem
  - Modem Firmware
  - WLAN Firmware

<table>
<thead>
<tr>
<th>CVE</th>
<th>References</th>
<th>Type</th>
<th>Severity</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-11940</td>
<td>A-79377832</td>
<td>N/A</td>
<td>Critical</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2254946</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2017-17772</td>
<td>A-72957385</td>
<td>N/A</td>
<td>High</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2150003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2018-11294</td>
<td>A-109741680</td>
<td>N/A</td>
<td>High</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2197481</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2018-5855</td>
<td>A-77527719</td>
<td>N/A</td>
<td>High</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2193421</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2018-11299</td>
<td>A-109741946</td>
<td>N/A</td>
<td>High</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2186953</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2018-11826</td>
<td>A-1111127853</td>
<td>N/A</td>
<td>High</td>
<td>WLAN HOST</td>
</tr>
<tr>
<td></td>
<td>QC-CR#2205957</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• Stability of Exploitation
• Conclusions
MBA and Modem images

- Modem Boot Authenticator
- mba.mbn
- modem.mdt
- modem.b00 – modem.b20
- Image format
Modem Secure Boot

1. PIL in Linux Kernel (Application CPU)
2. PBL in ROM (Modem CPU)
3. MBA (Modem CPU)
4. Modern image (Modem CPU)
5. DDR

- Load MBA and modem images to DDR
- Set modem registers, such as write MBA start address
- Reset modem CPU
- PBL set clock etc.
- Read and Auth MBA, Jump to MBA
- Read and Auth modem images
pil_boot

- The pil_boot function in Linux Kernel describes the boot flow of modem.
- Load mba.mbn, modem.mdt and modem.bxx to physical memory.
- Trigger MBA and modem images to be verified and run in Modem Processor.
- Linux Kernel can restart Modem Processor at any time, will hit pil_boot each time when restart.
Load modem.mdt, use the info to setup modem pa region

Call pil_msa_mss_reset_mba_load_auth_mdt to load msa.mbn and auth modem.mdt

Call pil_mss_mem_setup to register modem pa region to TZ

Call pil_assign_mem_to_subsys_and_linux to make Linux Kernel and MBA both can access the pa region

Call pil_load_seg to load modem.bxx to the pa region

Call pil_reclaim_mem to make Linux Kernel can't access the pa region any more

Call pil_msa_mba_auth to auth modem.bxx and start modem
TOCTOU Vulnerability

Modify the verified image
```c
@@ -640,24 +688,33 @@ int pill_boot(struct pill_desc *desc)
    goto err_deinit_image;
 }

- if (desc->subsys_vmid > 0) {
-    ret = pill_reclaim_mem(desc, priv->region_start,
-                          (priv->region_end - priv->region_start),
-                          desc->subsys_vmid);
-    if (ret) {
-        pill_err(desc, "Failed to assign %s memory, ret - %d\n",
-                 desc->name, ret);
-        goto err_deinit_image;
-    }
-    hyp_assign = false;
- }

    ret = desc->ops->auth_and_reset(desc);
    if (ret) {
        pill_err(desc, "Failed to bring out of reset\n");
        goto err_auth_and_reset;
    }

    pill_info(desc, "Brought out of reset\n");

+ if (nodes.dbg_cfg) { // just a switch can be set in userspace to enable our test
+     list_for_each_entry(seg, &desc->priv->segs, list) {
+         pill_modify_seg(desc, seg); // self defined function to modify segments
+     }
+ }

+ if (nodes.dbg_cfg) { 0} {
+     if (desc->subsys_vmid > 0) {
+         ret = pill_reclaim_mem(desc, priv->region_start,
+                                 (priv->region_end - priv->region_start),
+                                 desc->subsys_vmid);
+         if (ret) {
+             pill_err(desc, "Failed to assign %s memory, ret - %d\n",
+                      desc->name, ret);
+             goto err_deinit_image;
+         }
+         hyp_assign = false;
+     }
```
Debug Server Injection

Implementation

- Linux
  - Inject Shellcode
  - Command Queue
  - Result Buffer

- Shared Memory
  - Initialize Code
  - Demon Thread
    - Handle Commands
      - Read
      - Write
      - Breakpoint Condition

- Modem
  - HookPointA
    - C1000000 PUSH EAX
    - C1000000 JUMP Handler
    - C1000004 MOV
  - BreakPoint Handler
    - CheckCondition
    - Handle Commands
    - Exec Original Instructions
      - C1000000 PUSH EAX
    - Jump back HookPoint + 4
  - Dynamic Condition Codes
Agenda

• Introduction and Related Work
• The Debugger
• **Reverse Engineering and Attack Surface**
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• Stability of Exploitation
• Conclusions
Qualcomm WLAN

Application -> Android Framework -> Wifi Demon

Full MAC Layer

QCACLD2/3

Modem Firmware  WLAN Firmware

Linux Kernel

Baseband Subsystem
Hexagon
Qualcomm WLAN Architecture

Application → Android Framework → Wifi Demon

Full MAC Layer

QCACL2/3

Linux Kernel

WMI Handler

WLAN Firmware

Offload MAC Layer

Non-Data Handler → Data Handler

Physical Layer OTA Packet
Example - Management Beacon

- Full MAC Layer
- Offload Table
- Non-Data Handler
- QCACLD2/3
- Linux Kernel
- WLAN Firmware
- Wifi Demon
- Management Beacon
- Parse
- Forward
- Discard
- 80211 Management Beacon
- SSID
Firmware

• Modem load WLAN Firmware from /vendor/firmware/wlanmdsp.mbn
• IDA Disassembler
  • https://github.com/programa-stic/hexag00n/tree/master/ida
  • https://github.com/gsmk/hexagon
• Qualcomm SDK
  • https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools
• Instruction Reference
Reverse Engineering – Hint From Qualcomm

String Table

WMI Handler

drivers/staging/fw-api-fw/wmi_unified.h

```c
typedef enum {
    WM_INIT_CMDID = 0x1,
    /* Scan specific commands */
    WM_SCAN_CMDID = WM_GRP_STOP_CMDID, /* Full list of channels as defined by the regulatory that will be used by scanner */
    WM_SCAN_CHAN_LIST_CMDID, /* Overwrite default priority table in scan scheduler */
    WM_SCAN_CHAN_QIO_TLB_CMDID,
    /* This command to adjust the priority and min_max_rest_time */
    WM_SCAN_UPDATE_REQUEST_CMDID,
```
Reverse Engineering

• Targets To Reverse
  • WMI Handlers
    • Handle WMI commands from Linux Kernel
    • Send back WMI indication to Linux Kernel
  • Offload Handlers
    • Handle OTA Packets

Diagram:

- WMI Handler
  - Non-Data Handler
  - Data Handler

Offload MAC Layer

Physical Layer OTA Packet
WMI Handlers

drivers/staging/fw-api-fw/wmi_unified.h

```c
/**
 * Command IDs and command events
 */
typedef enum {
    /**< initialize the wlan sub system */
    WMICMDINITCMDID = 0x1,
    /**< Scan specific commands */
    WMISCANSPECIFICCMDID,
    /**< start scan request to FW */
    WMISTRANSCMDID = WMICMDGRPSTARTID(WMICMDGRPGROUPID),
    /**< stop scan request to FW */
    WMISTRANSTOPCMDID,
    /**< full list of channels as defined by the regulatory that will be used by scanner */
    WMICMDSCANCHANLISTCMDID,
    /**< overwrite default priority table in scan scheduler */
    WMICMDSCANCHSPRIOTBLCMDID,
    /**< This command to adjust the priority and min.max_rest_time */
    /**< of an ongoing scan request. */
    WMICMDSCANUPDATEREQUESTCMDID,
} WMICMDID;
```

```
LOAD: B0301D00  F4 69 02 B0
LOAD: B0301D04 01 30
LOAD: B0301D06 00 00 00 00
```

```
dd sub_B00269F4
```

```
dw 0x3001
```

```
dd 0
```
Offload Handlers
Sample Offload Handler

```
sub_B0004C2C:
  DataPtr = R17
  DataPtr1 = R21
  { call sub_B02859C4
    allocframe (0x30) }
  { R16 = R2 ; R19 = R0 }  
  { R2 = memw (R16 + 0) }  
  { R2 = memw (R2 + 0x10) }  
  { DataPtr = memub (R2 + 0x58)  
    R3 = memub (R2 + 0x59) }  
  { DataPtr |= asl (R3, 8)      
    R4 = memub (R2 + 0x58)  
    R5 = memub (R2 + 0x5A) }  
  { R5 |= asl (R4, 8) }  
  { DataPtr |= asl (R5, 0x10) }  
  { R3 = memub (DataPtr + 0) }  
  { R18 = and (R3, 0xF0)  
    R1 = and (R3, 0xC) ; management  
    if (!cmp.eq (r1.new, 0))  
      jump loc_B0004DF4 }  
  { P0 = cmp.eq (R18, 0x80)  
    DataPtr1 = memub (R2 + 0x5C)  
    R22 = memub (R2 + 0x5D) }  
  { if (P0)  
    jump loc_B0004C84 }  
```

OTA Packet Data Pointer

\[ [0x5B | 0x5A | 0x59 | 0x58] \]
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• **Vulnerability and Exploitation**
  • Escaping into Modem
  • Escaping into Kernel
• Stability of Exploitation
• Conclusions
The Roadmap
## Mitigation Table (WLAN & Modem)

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap ASLR</td>
<td>Y</td>
</tr>
<tr>
<td>Heap Cookie</td>
<td>Y</td>
</tr>
<tr>
<td>Stack Cookie</td>
<td>Y</td>
</tr>
<tr>
<td>W^X</td>
<td>Y</td>
</tr>
<tr>
<td>FRAMELIMIT*</td>
<td>Y</td>
</tr>
<tr>
<td>FRAMEKEY**</td>
<td>Y</td>
</tr>
<tr>
<td>Code &amp; Global Data ASLR</td>
<td>N</td>
</tr>
<tr>
<td>CFI</td>
<td>N</td>
</tr>
</tbody>
</table>

*FRAMELIMIT Register* - If SP < FRAMELIMIT throw exception

**FRAMEKEY Register** - Return Address XOR FRAMEKEY. A random integer different for every thread
The Vulnerability (CVE-2019-10540)
The Vulnerability (CVE-2019-10540)

Copy items from packet into Global Static Buffer.
Max Item Count = 10
Send 11 items -> Overflow!

```
[GLOBAL] char *GlobalBuffer[10 * 0xB0 + 6];

unsigned int itemCount = 0;
for (unsigned int i = 0; i < Length; i += 0x44) {
    memcpy (GlobalBuffer + 6 + itemCount * 0xB0, 
             OTA_DataPtr + i, 
             0x44);
    itemCount++;
}
```

* Translated and simplified the code flow
Data & Address of Overflow

Global Buffer

- Head (6 bytes)
- Item 1 (0xB0 bytes)
- Item 2 (0xB0 bytes)
- ...
- Item 9 (0xB0 bytes)
- Item 10 (0xB0 bytes)
- Overflow (0x44 bytes)
- (0xB0 – 0x44 bytes)
- Overflow (0x44 bytes)
- (0xB0 – 0x44 bytes)
- ...

+0xB0 * 10
Smart Pointer Around Overflow Memory

Global Buffer

- Head (6 bytes)
- Item 1 (0xB0 bytes)
- Item 2 (0xB0 bytes)
- ...
- Item 9 (0xB0 bytes)
- Item 10 (0xB0 bytes)

Overflow (0x44 bytes)
- (0xB0 – 0x44 bytes)
- Overflow (0x44 bytes)
- (0xB0 – 0x44 bytes)
- ...

+0
0x00000000

+4
0x00000000

+8
0x00000000

+C
SmartyPointer
Usage Of Smart Pointer

Char **AddressOfSmartPointer = GlobalBuffer + 6 + 0xB0 * 11 + 0xC;
char *SmartPointer = *AddressOfSmartPointer;
char *MacAddress = OTA_DataPtr + 0x10;
char *BYTE_C = OTA_DataPtr + 0x10 + 0x20;
char *BYTE_D = OTA_DataPtr + 0x10 + 0x21;
char *BYTE_14 = OTA_DataPtr + 0x10 + 0x22;
if (TestBit(SmartPointer, 0) == 1) {
    if (memcmp(SmartPointer + 6, MacAddress, 6) == 0) {
        *(SmartPointer + 0xC) = *BYTE_C;
        *(SmartPointer + 0xD) = *BYTE_D;
        *(SmartPointer + 0x14) = *BYTE_14;
    }
}

* Translated and simplified the code flow
Usage Of Smart Pointer

Char **AddressOfSmartPointer = GlobalBuffer + 6 + 0xB0 * 11 + 0xC;
char *SmartPointer = *AddressOfSmartPointer;  // ← Overwrite with vulnerability
char *MacAddress = OTA_DataPtr + 0x10;
char *BYTE_C = OTA_DataPtr + 0x10 + 0x20;
char *BYTE_D = OTA_DataPtr + 0x10 + 0x21;
char *BYTE_14 = OTA_DataPtr + 0x10 + 0x22;
if (TestBit(SmartPointer, 0) == 1) {  // ← The only constraint, Bit0 == 1
    if (memcmp(SmartPointer + 6, MacAddress, 6) == 0) {
        // ← From OTA Data, could be bypass
        *(SmartPointer + 0xC) = *BYTE_C;  // ← Overwrite 0xC
        *(SmartPointer + 0xD) = *BYTE_D;  // ← Overwrite 0xD
        *(SmartPointer + 0x14) = *BYTE_14;
    }
}

* Translated and simplified the code flow
Global Write With Constraint

Step 1 Overwrite SmartPointer

Step 2 Global Write (Using SmartPointer)

SmartPointer

0xFFFFFFFF

0xFFFFFFFF
+4 00 00 00 00
+8 00 00 00 00
+C 12 34 56 78

00 00 00 01
+0 00 00 00 01

+4 MA CA 00 00
+4 MAC
+8 AD DR ES SS
+8 Write
Global Write With Constraint

How to write 4 bytes?

**Step 1 Overwrite SmartPointer**

**SmartPointer**

0xXXXXXXXX

**Step 2 Global Write (Using SmartPointer)**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>00 01 00 01</td>
</tr>
<tr>
<td>+4</td>
<td>MACA 00 00</td>
</tr>
<tr>
<td>+8</td>
<td>AD DR ES SS</td>
</tr>
<tr>
<td>+C</td>
<td>12 34 ?? ??</td>
</tr>
</tbody>
</table>

**Write Low 2 Bytes**

**Step 3 Overflow SmartPointer**

**SmartPointer**

0xXXXXXXXX+2

**Step 4 Global Write (Using SmartPointer)**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>00 01 00 01</td>
</tr>
<tr>
<td>+4</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>+8</td>
<td>MA CA DD RE</td>
</tr>
<tr>
<td>+C</td>
<td>?? ?? SS SS</td>
</tr>
</tbody>
</table>

**Write High 2 Bytes**
Global Write With Constraint
The Bit0 != 1?

- C
  00 00 00 01

-8
  00 00 00 00

-4
  00 00 00 01

+0
  00 00 00 00

+4
  MACA 00 00

+8
  AD DR ES SS

+C
  12 34 ?? ??

Bit0 == 1?
MAC

Bit0 != 1
MAC

Target
## Control PC & R0

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0x00010000</td>
</tr>
<tr>
<td>+04</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+08</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+0C</td>
<td>0x00000001</td>
</tr>
<tr>
<td>+10</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+14</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+18</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+1C</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+20</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+24</td>
<td>0x12345678(PC)</td>
</tr>
<tr>
<td>+28</td>
<td>0x87654321(R0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00</td>
<td>0x00010000</td>
</tr>
<tr>
<td>+04</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+08</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+0C</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+10</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+14</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+18</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+1C</td>
<td>0x00010001</td>
</tr>
<tr>
<td>+20</td>
<td>0x00000000</td>
</tr>
<tr>
<td>+24</td>
<td>TARGET PC</td>
</tr>
<tr>
<td>+28</td>
<td>TARGET R0</td>
</tr>
</tbody>
</table>
Transform To Arbitrary Write

TARGET PC

TARGET R0

Item1
Payload1
Item2
Payload2
...

FOP Gadget

LOAD:B000DBD8 24012300
LOAD:B000DBDC04012282
LOAD:B000DBE000409F52
LOAD:B000DBE4030002A2

{R3 = memw (R2 + 0) ; R4 = memw (R2 + 4)}
{R2 = memw (R2 + 8) ; memw (R0 + 4) = R4}
{jump R31
memw (R0 + 8) = R2 ; memw (R0 + 0) = R3}
Run Useful FOP Gadget

Step 1 Arbitrary Write  Overwrite function pointer →

Step 2 Arbitrary Write  Overwrite data pointer →

Step 3 Send payload packet and trigger the PC →

Function Pointer (PC)

Data Pointer (R0)

Item1
Payload1

Item2
Payload2

...

## Memory Mapping RWX

### CreateMapping(args, ⋯)

- **R0 = 0x42420000** Virtual Address
- **R1 = 0x936a0000** Physical Address
- **R2 = 0x1000** Size
- **R3 = 4** Unknown
- **R4 = 7** Permission RWX
Memory Mapping RWX

TARGET PC

TARGET R0

Item1
Payload1

Item2
Payload2

...
Copy Shellcode to 0x42420000

**Step 1 Arbitrary Write** Overwrite function pointer

**Step 2 Arbitrary Write** Overwrite data pointer

![Diagram showing the flow of steps and variables]
Trigger Shellcode

Step 1 Arbitrary Write Overwrite function pointer →

0xB0000020 (PC)

Any Value (R0)

Step 2 Trigger

0xB000020
Shellcode...

...
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• Stability of Exploitation
• Conclusions
The Roadmap
From WLAN to Modem

<table>
<thead>
<tr>
<th>Actions From WLAN</th>
<th>Eligible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Set*</td>
<td>N</td>
</tr>
<tr>
<td>Write Modem Data</td>
<td>N</td>
</tr>
<tr>
<td>Call Modem Complex Function**</td>
<td>N</td>
</tr>
<tr>
<td>Call Modem Simple Code Snippet***</td>
<td>Y</td>
</tr>
<tr>
<td>Map Modem Memory</td>
<td>Y</td>
</tr>
</tbody>
</table>

* TLB is a Hexagon Instruction to modify the Memory Page Attribute
** Complex Function uses the resource of Modem, or calls System Call
*** Simple Code Snippet mean code has only register operation
Map Modem Memory into WLAN

WLAN Process Virtual Address

Virtual Address

Physical Address
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• Stability of Exploitation
• Conclusions
The Roadmap
The Attack Surfaces

- Userspace APP
- Linux Kernel
- TrustZone
- Modem
- AT Command
- Glink
- DIAG
- QMI
- WMI
- APR
- Share Memory
We’ve found an arbitrary memory read/write vulnerability that could bypass all the mitigations of Linux Kernel. From Modem into Linux Kernel.

In these attack surfaces,

But we are unable to disclose the detail now.
Agenda

• Introduction and Related Work
• The Debugger
• Reverse Engineering and Attack Surface
• Vulnerability and Exploitation
• Escaping into Modem
• Escaping into Kernel
• **Stability of Exploitation**
• Conclusions
Deliver the Payload Over-The-Air

Packet Losing Rate 90%+!
Deliver the Payloads Using Pixel2
The Roadmap
Demo
Future Works

• There are still lots of mystery in the WLAN.
  • We were only reversed a small part of the code
  • Lots of functions are unknown

• How to fuzz the WLAN Firmware?
  • Reverse engineering is quite…
  • How to fuzz closed source target and Hexagon architecture effectively?

• Translate Hexagon Instruction to C?
  • IDA/Ghidra F5 plugin?
Timeline

• 2019-2-14 Find the Modem debug vulnerability on MSM8998
• 2019-3-24 Find the WLAN issue and report to Google
• 2019-3-28 Google forwards the issue to Qualcomm
• 2019-4-24 Google confirms the WLAN issue as Critical
• 2019-5-08 Report the WLAN into Linux Kernel issue to Google
• 2019-5-24 Google confirms the WLAN into Linux Kernel issue
• 2019-5-28 Submit the full exploit chain to Google
• 2019-6-04 Google reply unable to reproduce the full exploit chain
• 2019-6-17 Improve the stability and submit to Google
• 2019-7-19 CVE Assigned by Google
• 2019-7-20 Qualcomm confirms issues will be fixed before October
• 2019-8-0? Google release the fix for Google Pixel2/Pixel3
THANK YOU

https://blade.tencent.com