Exploiting Qualcomm WLAN and Modem Over the Air

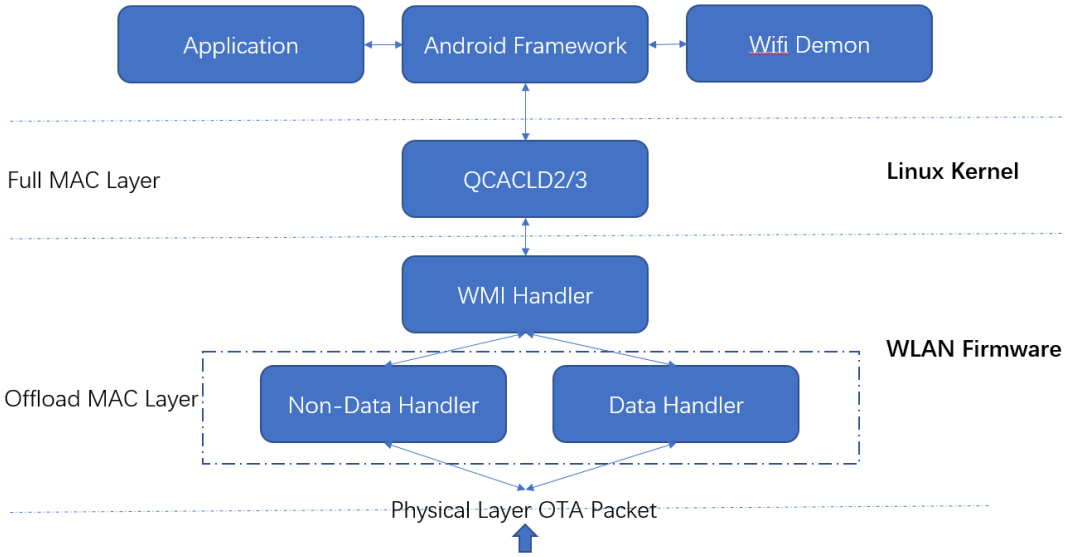
Xiling Gong of Tencent Blade Team

Part I – WLAN

1 The Vulnerability

1.1 Background

Here is a simple figure for WLAN architecture of Qualcomm.



From the figure we can see that, Qualcomm WLAN firmware is not a *FULL MAC* firmware. The studies on Broadcom[1] and others show that, firmware of other vendors implement the full functions of the WLAN/WIFI stack. But Qualcomm is different, most of the MAC functions of WLAN/WIFI are implemented in the Linux Driver called QCACLD. The firmware of Qualcomm is much simpler. Thus leading to a smaller attacker surface in the firmware.

The main attack surface I think is in the Offload Handler. Offload Handler will inspect the OTA packet and if is interested, will parse the packet and do some job accordingly. The Offload Handler is a table, with some different handlers according to the status of the WIFI.

The incoming OTA packets will be first processed by the Offload Handlers in the table. And Offload Handler will decide the next process, parsing, discarding, send to Linux, etc.

Give an example for the Management Beacon Frame handler. When the userspace application turn on the WIFI, the Android framework will issue command to the QCACLD driver, and finally WLAN firmware will register a handler into the Offload Handler Table. When the two handlers encounter interesting packet, such as Management Beacon Frame, it will parse the incoming packet to check whether the packet contains an interesting SSID and if found will try to connect to it automatically. Otherwise the handler will simply transfer the packet to the next handler.

1.2 The Mitigations in WLAN and Modem

Let's first check the mitigation status of WLAN to setup an exploitation strategy.

Mitigation	Status
Heap ASLR	Y
Heap Cookie	Y
Stack Cookie	Y
Code & Global Data ASLR	N
W^X	Y
CFI	N

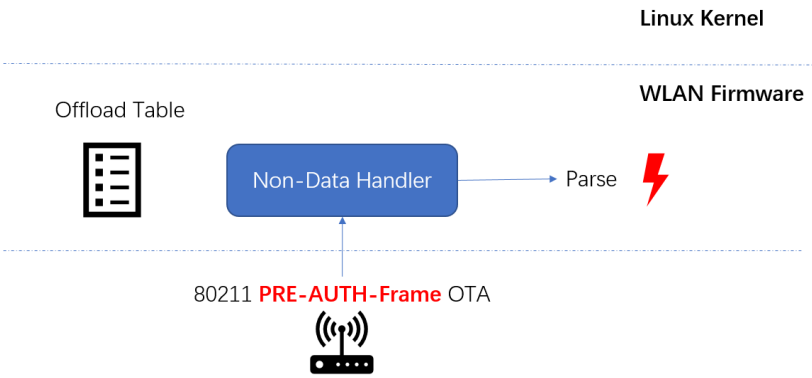
From the mitigation table above in WLAN and Modem, we know that W^X protection is on, so we are unable to jump the PC to the payload on the heap or data area directly. Also, we are unable to write to code segment directly. If we want to run arbitrary code, we should change the page table attribution or employ some functions like *mprotect* to do the job.

The Stack Cookie is enabled, so we are unable to use traditional ROP on the stack. Luckily, CFI is not on, so we are able to use the technique called Function Oriented Programming (FOP). You'll see late in the exploitation, that all the gadget in the FOP is constructed by a series of function call.

The heap has two mitigations. Heap ASLR will random the address space of heap and heap cookie will protect each memory block allocated by a head with a random cookie. These two mitigations make exploitation from heap overflow pretty difficult.

1.3 The Vulnerability(CVE-2019-10540)

The vulnerability occurs when one of the Offload Handler handles one of the pre-auth-frame of 80211. The PRE-AUTH means, you don't have to connect to a specified hotspot or access point! Just turn on the WIFI will lead to vulnerable.



Please note, when we write and submit this white paper, the issue is still not been fully fixed. Although we are unable to disclose the exact address of the vulnerable code for responsible disclosure, we'll talk about the key principle of the vulnerability, that is enough for you to understand the issue.

If you are interesting about the detailed information of the vulnerability, please check the Android Security Bulletin August 2019.

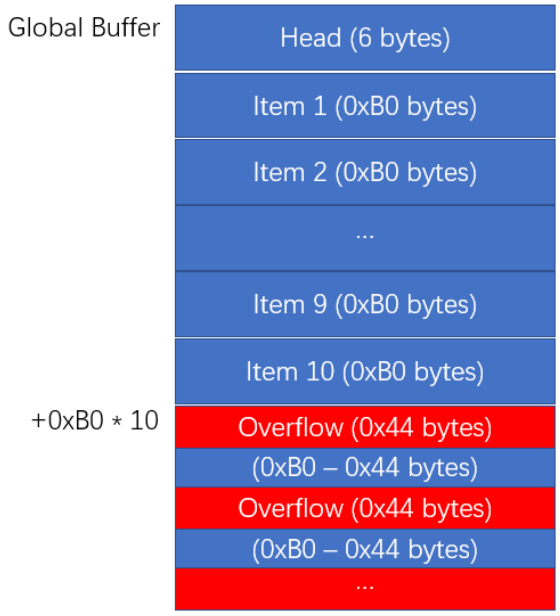
I've translated and simplified the assembler code into C as the following code snippet.

```
[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++;
}
```

You can see from the code, the GlobalBuffer contains 10 entries. But the loop will copy as much of data as possible to the GlobalBuffer. Where is the buffer? It is in the Global Static Data segment. There is no check on the data count. Send 11 items will cause a global buffer overflow.

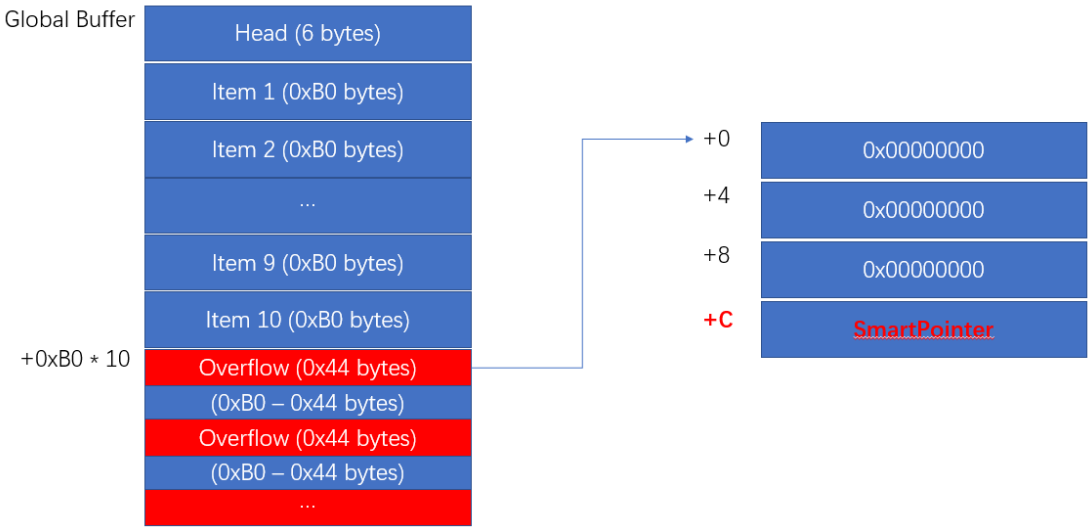
From the vulnerability code, we know that each loop will copy 0x44 bytes. Trigger the overflow, we can write up to 0x44 arbitrary bytes of data to the location start from GlobalBuffer. Because there is no ASLR on the Global Data Segment, so the overflow address is fixed and the data usage around the overflow address is also fixed. That's quite convenient for the exploitation.



2 The Exploitation

2.1 Overflow to Global Write with Constraint

Analyze the data near the overflow address, we can quickly find that, there is a very good pointer at offset 0xC. We call it SmartPointer which point to a very useful data.



Let's check the usage of the SmartPointer. To better understanding the code, I've tranlated the code to C and simplified them as the following figure.

```
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;
        }
}
```

From the code, we can see that, if we modify the SmartPointer to a destination, it will do some tests, and if the test pass, we'll write arbitrary code to the destination. The detail is that, it will first test bit0 of the destination. If pass, then it will compare the memory at SmartPointer to the MAC address of the sender. If the memcmp also pass, then it will write two bytes to the offset 0xC, 0xD of SmartPointer.

That means, if a destination we are interesting satisfy the two conditions, we can write two arbitrary bytes to the destination. So we get an global arbitrary write primitive, though the primitive has two constraints.

Set SmartPointer = Destination = X, If

(1), bit 0 of X == 1,

(2), byte of X[6:12] == MAC address of sender

Then we can write arbitrary bytes to

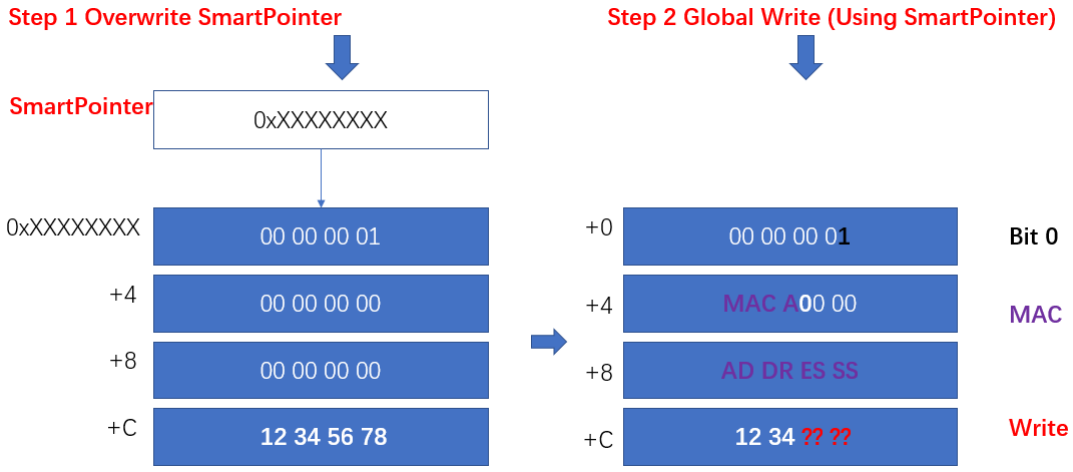
X + [0xC, 0xD, 0x14]

Check the two constraints carefully, we can quickly find that, the condition (2), the MAC address of the sender, it is controlled by ourselves. So, in fact we can surely make our packet satisfy this condition. So only condition (1) bit0 == 1 is a real constraint.

To summarize, I've drew another figure for the code.

```
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, MacAddress, 6) == 0) {    // ← From OTA Data, could be bypass
                *(SmartPointer + 0xC) = *BYTE_C;  // ← Overwrite 0xC arbitrary
                *(SmartPointer + 0xD) = *BYTE_D;  // ← Overwrite 0xD arbitrary
                *(SmartPointer + 0x14) = *BYTE_14;
        }
}
```

And the following figure is a simple illustration about the global write with constraint.



There are in fact 2 steps. The first step will send a packet to trigger the overflow and overwrite the content of SmartPointer. The second step will send a packet to manipulate the new SmartPointer, to compare and (if pass) write to the new destination.

2.2 Control PC and R0

Although has some constraint, the global write is pretty powerful. Let's see how we utilize the global write to control the PC and R0.

The basic method here is, review the memory space and find out a meaningful place where satisfy the condition and reachable by our remote packet. There are two tricks

(1) How to write 4 bytes. If bit0 and bit16 of X are both 1, then X + [0xC, 0xD, 0xE, 0xF] could all be write. That's 4 bytes

(2) If bit0 of X is 1, then X + [0xC] + [0xC] + ... are all writable. We first write 1 to X + 0xC, then write X + 0xC + 0xC, and go on.

Now we are able to write 4 bytes, and write to some place we are interesting but not satisfy the condition.

The next step is to use the global write to control PC and R0. That's not quite difficult. Analyze the memory status, find a function pointer and a data pointer, modify them, then we are controlling the PC and R0.

The following figure illustrate the procedure. The SmartPointer will be moved from top to bottom to make our destination +24 & +28 satisfy the constraint.

Address	Value
00	0x00010000
+04	0x00010001
+08	0x00000000
+0C	0x00000001
+10	0x00000000
+14	0x00000000
+18	0x00000000
+1C	0x00000000
+20	0x00000000
+24	0x12345678(PC)
+28	0x87654321(R0)

Address	Value
+00	0x00010000
+04	0x00010001
+08	0x00000000
+0C	0x00010001
+10	0x00010001
+14	0x00000000
+18	0x00010001
+1C	0x00010001
+20	0x00000000
+24	TARGET PC
+28	TARGET R0

SmartPointer

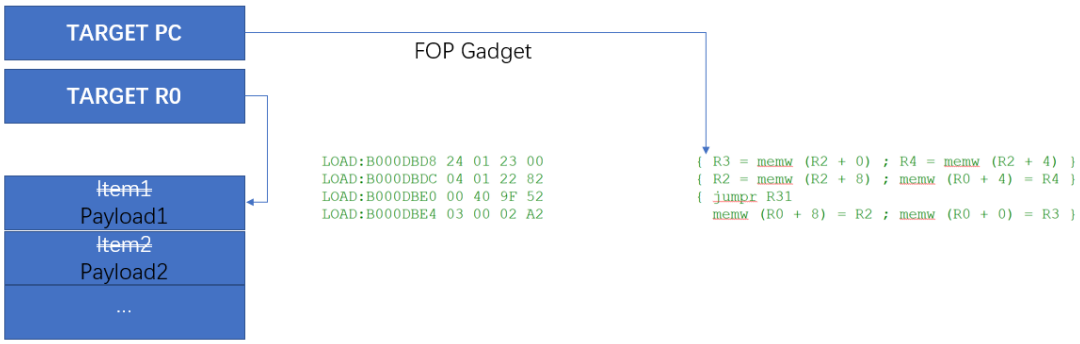
After controlled the PC and R0, we are able to run some Function Original Gadget (FOP) to transform the write to an arbitrary write without constraint. Also, we are able to do other complex tasks such as modify the page attribute to RWX and trigger shellcode.

2.3 Transform to Arbitrary Write

The Global Write we have is still not very convenient, because the place we want to write should have the lowest bit equals 1. Generally, we'd better transform it to an arbitrary write. As we've controlled the PC and R0, we can trigger a FOP to do the job.

Setup the PC and R0, and prepare the payloads at the known address, then send a packet to trigger the FOP, we'll get the arbitrary write we want.

Here is a simple illustration figure.

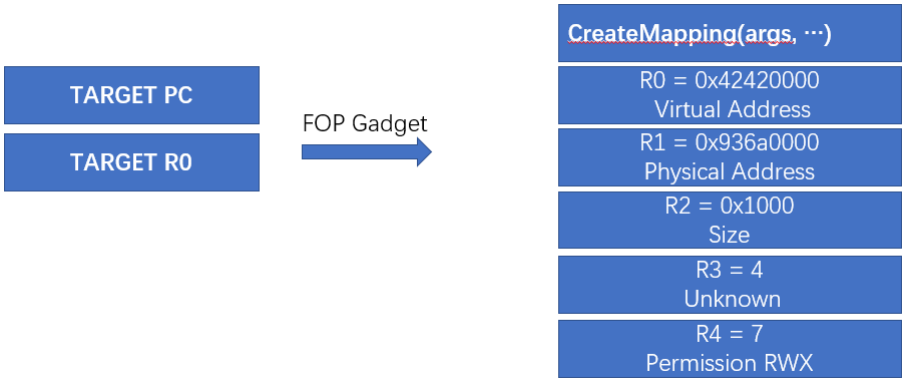


2.4 Do Memory Mapping

Now let's bypass the W^X protection, by change the attribution of the page table.

I don't know where the page table is in the memory, so I'm unable to write to the page table directly using arbitrary write. Luckily, I found the function like mprotect, ***the mapping create function***, that maps arbitrary physical page to arbitrary virtual address with specified permission. Calling this function allow us to change the permission of pages.

Before we go, we should know that this function need 5 parameters. So we have to use some FOP gadget to setup the registers and then finally trigger the mapping create function.

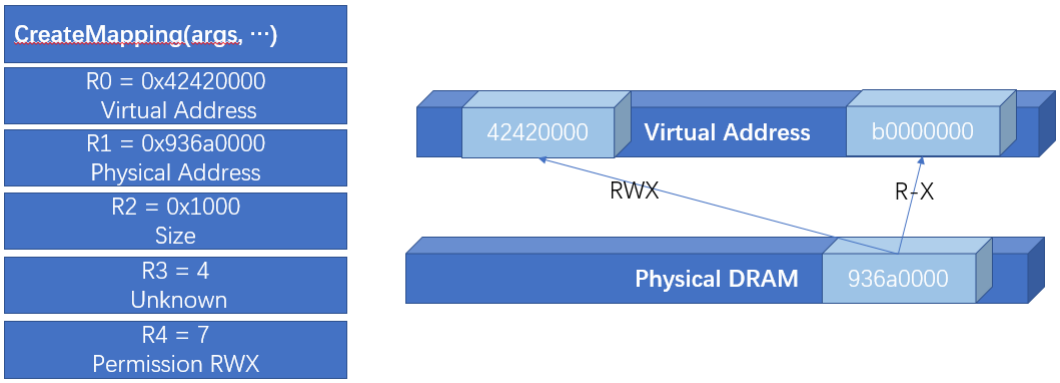


Finally, we have remapped

0xB0000000 (with physical address 0x936a0000)

To

0x42420000.



2.5 Copy the shellcode to 0x42420000

Now things are simple, we can do arbitrary write and we have a page that is RWX, there are lots of choices. The method I use is a bit tricky. I first write a code snippet, which could do memory copy to the address 0x42420020, that is

0x42420020 0xa09dc000 -> allocframe(#0)
0x42420024 0x5a00d266 -> call memcpy
0x42420028 0x961ec01e -> dealloc_return

And then overwrite one of the non-data handler to target 0xB0000020 with R0 set to 0x42420000.

0xb0091320 -> 0xB0000020
0x00000000 -> 0x42420000

So now send the packet will finally jump to 0xB0000020 and then copy the packet to 0x42420000. You know that 0xB0000000 is actually of the same physical memory with 0x42420000. So, after this memcpy finishes,

0x42420020 0xa09dc000 -> allocframe(#0)
0x42420024 0x5a00d266 -> call memcpy

The original instruction

0x42420028 0x961ec01e -> dealloc_return

will be replaced by the new shell code in the incoming OTA packet.

Now we are running arbitrary code in the WLAN.

Part II – WLAN into Modem

Starting from Snapdragon 835, the WLAN is integrated into the Modem subsystem. Does compromise WLAN also compromise Modem? What is the relationship between WLAN and Modem?

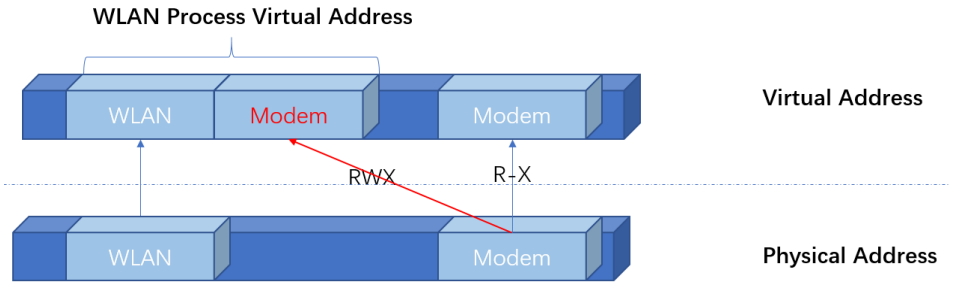
After we exploited WLAN, we found there are some constraints to WLAN, that is, WLAN is unable to access all the resource Modem owns. For example, call some system calls or functions will cause the Modem crash. So, we guess the WLAN is an isolated process of Modem. We need to break into Modem from WLAN. The following table shows what can we do from WLAN.

Actions From WLAN	Eligible?
TLB Set	N
Write Modem Data	N
Call Modem Complex Function**	N
Call Modem Simple Code Snippet*	Y
Map Modem Memory	**Y**

You can see that, if we call complex Modem Functions from WLAN, that is non-eligible, the Modem subsystem will crash. Complex Function mean a function that uses the resource of Modem (such as system calls, file system). If we using the TLB instruction to change the attribution of the memory that owns by Modem, it will also crash.

For simple functions that just simply uses the operate the registers, it's OK to call from WLAN. But this is useless. Finally, I found that WLAN is able to map the memory of Modem, like WLAN map the memory of its own!

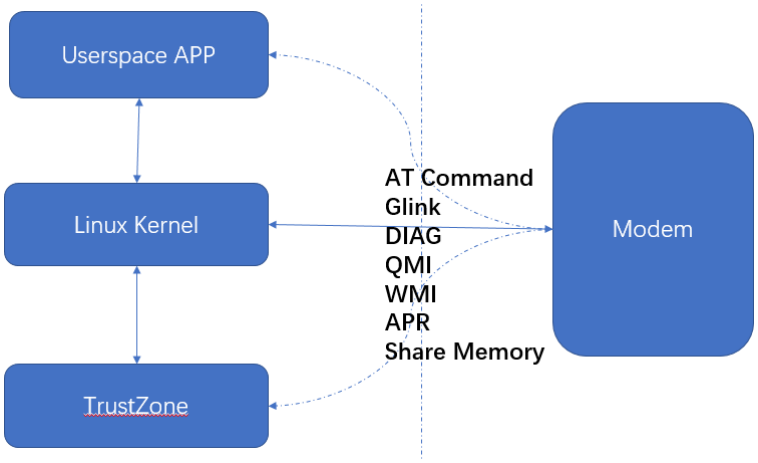
So, the solution is, just simply map the code memory of Modem into WLAN and then modify the code directly. So that we are running code in Modem.



And now we know that, since WLAN could map the memory of Modem, if we've compromised WLAN, we've also compromised Modem. It's good news I think, because there is no public exploitation on Qualcomm Modem since the year 2013, it's not easy. Now we have another choice that may be a shortcut if we are targeting the Qualcomm Modem^^

Part III – From Modem into Linux Kernel

The baseband subsystem is on a separate chip, not on the main application processor that Linux Kernel running. There are lots of interaction between baseband subsystem and application processor. Each of the interface is an attack surface.



For example, there are many WLAN Host issues in the Android Security Bulletin every month. Most of them are firmware to Linux Kernel vulnerabilities. These issues could be used to escaping from the baseband into Linux Kernel. And there are also potential attack surfaces from Modem into the high privileged userspace process and TrustZone through relay of the Linux Kernel.

In these attack surfaces, we've found a prefect vulnerability that could arbitrary memory read/write the memory of Linux Kernel from Modem. We'll disclose this issue in the future when it has been fixed.

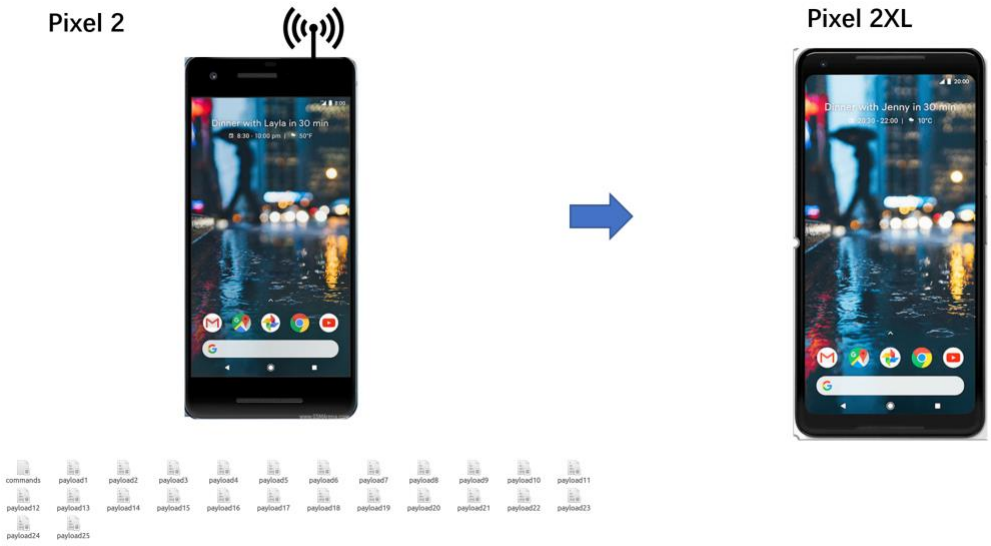
Part IV – Stability of The Exploitation

When I start my work, I'm using Scapy[2] along with a WIFI dongle in Monitor Mode to send packets. As I test, the packet losing rate is very high when I send the packet, so I have to send the packet repeat, maybe one packet 100 times to make sure that the target receives the packets. The first exploitation is quite a long journey, about 36 packets in total. Although finally I've shrank the packet count to 25, the packet losing is still a big problem. Losing one packet will causes the exploitation fail. The success rate is quite low.

The reason why losing packet is quite complex for me. Finally, I found the solution. I have a Google Pixel2 with Snapdragon inside, which could send and receive packets normally. Surely the Pixel2 could solved the problem I encounter and send packet correctly. So how about re-use the function that send packet in a Pixel2?

Hook the firmware function which send specified packets we want on a Pixel2, replace the content of the packet to our attack payload. That's the solution. Finally, we are using a Pixel2 to attack the Pixel2XL. You may ask how to hook the firmware function. I'm actually using the Modem deugger vulnerability which works on MSM8998. I'll talk about this issue

when it has been fixed.



References

1 https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

2 https://scapy.net/