

Marco Grassi

Muqing Liu

Tianyi Xie



Exploitation of a Modern Smartphone Baseband

9.Aug.2018

Agenda

Introduction and Related Work

Background

Baseband Remote Code Execution from Mobile
Pwn2Own: A Case Study, Exploiting the Huawei
Baseband

Delivering the Exploit

Exploitation

Demo

Escaping the Baseband

Conclusions



About Us

Members of Tencent KEEN Security Lab (formerly known as KeenTeam)

Marco Grassi (@marcograss):

- My main focus now is hypervisor, baseband, firmware. But sometimes I go back to iOS/Android/macOS and sandboxes etc.
- pwn2own 2016 Mac OS X Team, Mobile pwn2own 2016 iOS team, pwn2own 2017 VMWare escape team, Mobile pwn2own 2017 iOS Wi-Fi + baseband team

Tianyi Xie:

- CTF player, captain of CTF Team eee and A*0*E.
- Champion of CODEGATE CTF 2015 as member of team 0ops.
- Pwn2Own 2017 VMware escape team, Mobile Pwn2Own 2017 baseband team.

Muqing Liu

- CTF player, member of Team eee and A*0*E
- Mobile Pwn2Own 2017 baseband team



About Tencent Keen Security Lab

White Hat Security Researchers

Several times pwn2own winners

We are based in Shanghai, China

Our blog is <https://keenlab.tencent.com/en/>

Twitter @keen_lab

Research area:

- PC security: Browser, Sandbox, Kernel (Windows, Linux, MacOS)
- Mobile security: Mobile Browser, Mobile sandbox, Mobile kernel (Android, iOS)
- Baseband and firmware
- Virtualization: VMWare, Hyper-V, XEN, QEMU
- Car research: Tesla, BMW
- App security



Introduction And Related Work

There is a relatively small amount of public research on Basebands

The complexity is quite high, having to deal with very complex specifications, which means also a higher entry barrier, since you need to know the topic at least a little bit.

We have billions of smartphones in the world and most of them have a Baseband processor.

Basebands can provide a first RCE bug triggered over the air to compromise a smartphone without user interaction.

Related Work

At Mobile Pwn2Own 2017 we successfully exploited the Huawei baseband, so our showcase and analysis will be on that baseband.

Other works on basebands:

- Comsecuris – Breaking Band (Samsung Shannon Baseband)
- Amat Cama – A Walk with Shannon (Samsung Shannon Baseband)
- Comsecuris - There's Life in the Old Dog Yet (Intel Baseband)
- Guy – From 0 to Infinity (Intel Baseband)
- Muiruri, Artenstein, Dorfman – The Baseband Basics (MTK Baseband)
- Ralf-Philipp Weinmann – Baseband Attacks and other work (Qualcomm Baseband)
- There are also other resources omitted for space constraints (sorry!)

A Modern Smartphone Architecture

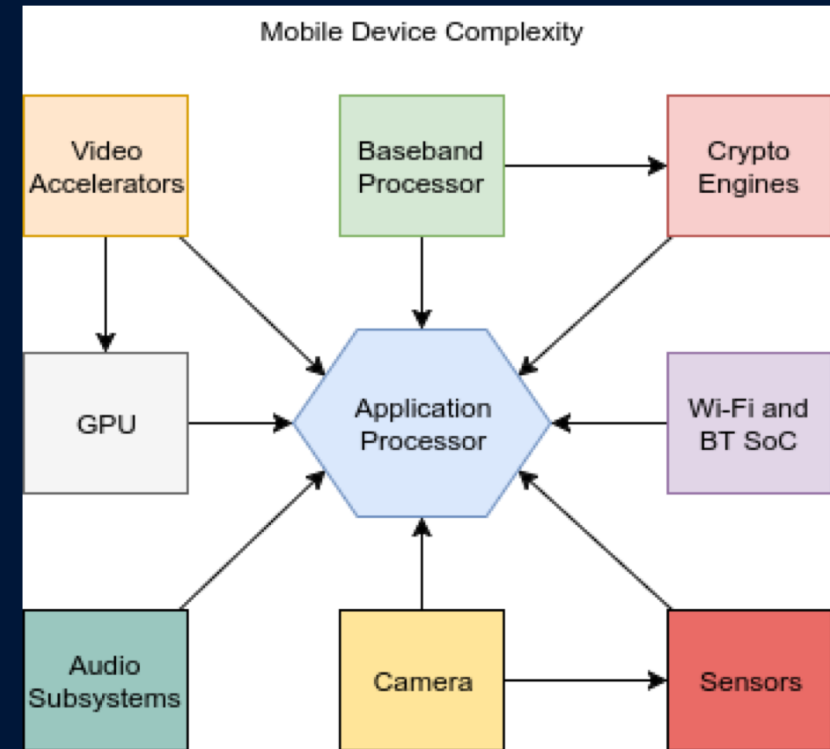
A modern smartphone is not a single CPU executing an OS anymore.

Several other processors involved in the radio area:

- Baseband processor
- Wi-Fi and Bluetooth SoC

The baseband handles the radio communication with many types of networks: 2G, 3G, 4G etc.

Those radio components can be attacked remotely



Source:

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

A Modern Smartphone Architecture 2

The AP runs your OS and apps (Android)

The Baseband runs a RTOS

They communicate with

- USB
- PCI-e
- Shared Memory
- SDIO
- ...

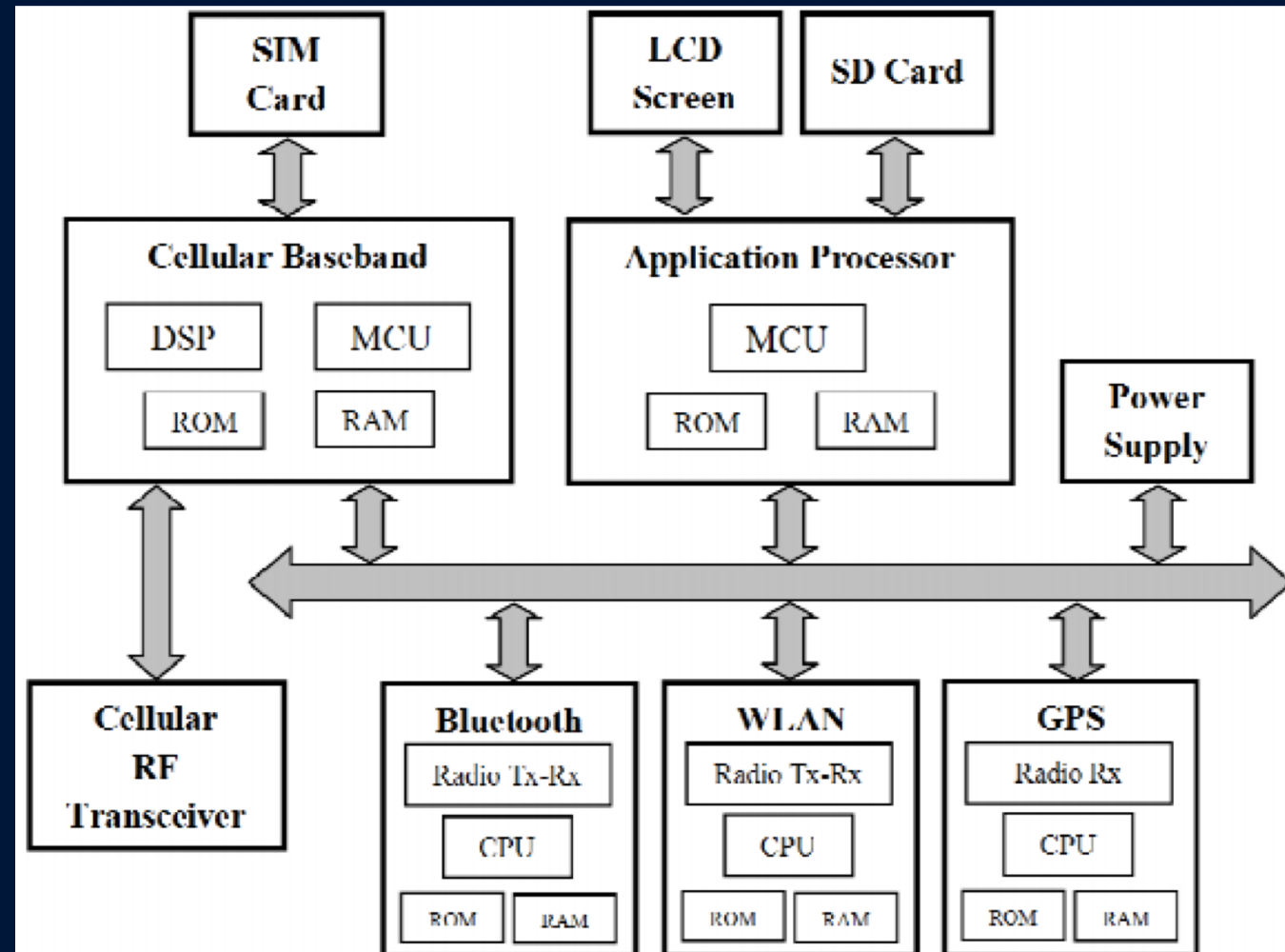
Separate systems. If you get RCE you run code on the Baseband, not the AP.

The Basebands lags behind in terms of mitigations compared to the AP

In Huawei we noticed they lack of ASLR and even stack cookies, making the remote exploitation easier.

This lack of mitigations is widespread between all manufacturers.

Source : <https://www.evelta.com/introduction-smartphone-architecture>



Why target the baseband

It has several advantages:

1. Less understood and less audited attack surface
2. It can be exploited remotely without user interaction, potentially from long distances
3. Lack of mitigations compared to a Modern mobile OS
4. Often the device manufacturer doesn't have access to the baseband code. They cannot easily audit it.
5. Complexity.

Radio technologies

Nowadays Radio technologies made great improvements thanks to SDRs (Software Defined Radios).

They allow researchers to communicate with basebands and setup fake Base Stations.

They are affordable.

BladeRF, LimeSDR, USRP...

However opensource implementations can only cover SOME of the baseband radio stacks. We will see later why this will turn into a problem for us.

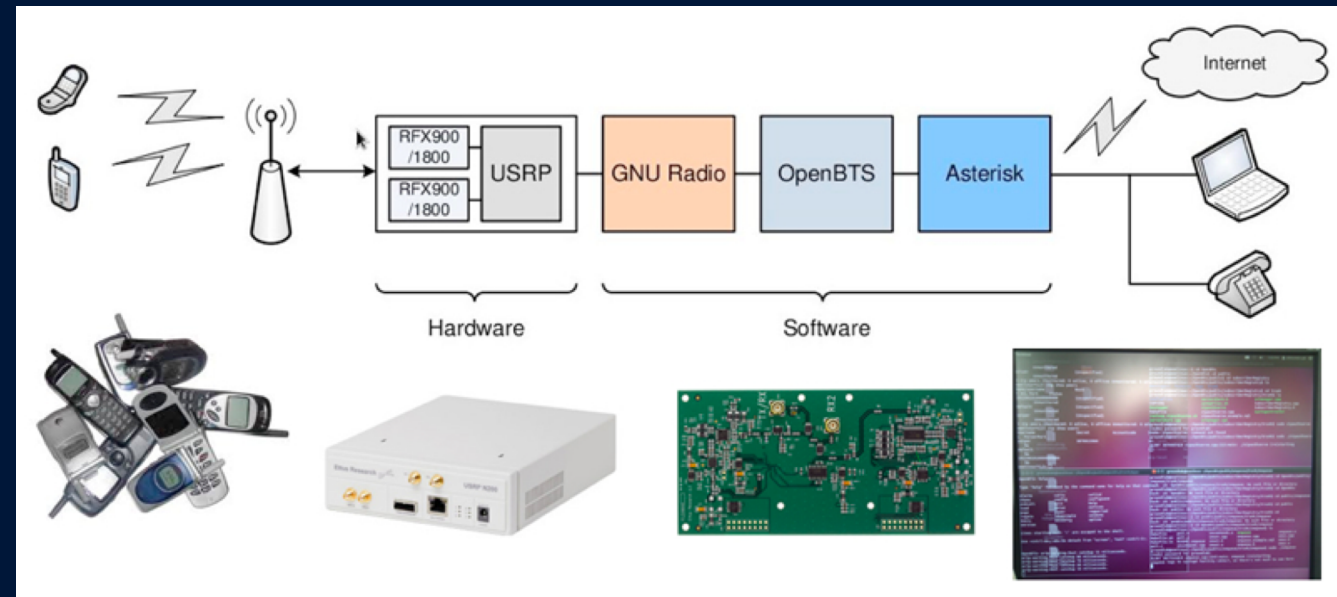
How do we attack a baseband?

The «traditional» approach is to fake a base station with a SDR.

The old networks blindly trust a fake base station (in newer networks there is mutual authentication)

We can then send malformed messages and exploit the Baseband.

In our case it was not so simple as we will see.



CDMA

Our showcase bug will be in CDMA, so you need at least to know what CDMA is.

CDMA (in the context of cellular network) is: " a family of 3G mobile technology standards for sending voice, data, and signaling data between mobile phones and cell sites. "

It was a competing technology of UMTS in different part of the world.

Probably in USA you know what it is anyway, since you use it.

Background

Basebands are basically black boxes running a firmware on a separate CPU inside your smartphone

They are similar to some IoT devices, they run a Real Time Operating Systems, with lot of tasks. Each responsible for some layer or component.

The complexity is huge. You can get an idea by checking the specifications of Layer 3 3GPP TS 04.08, which consists of hundreds of pages, and covers only 1 layer of 1 radio network!

The Baseband RtOS

RTOS stands for Real Time Operating Systems

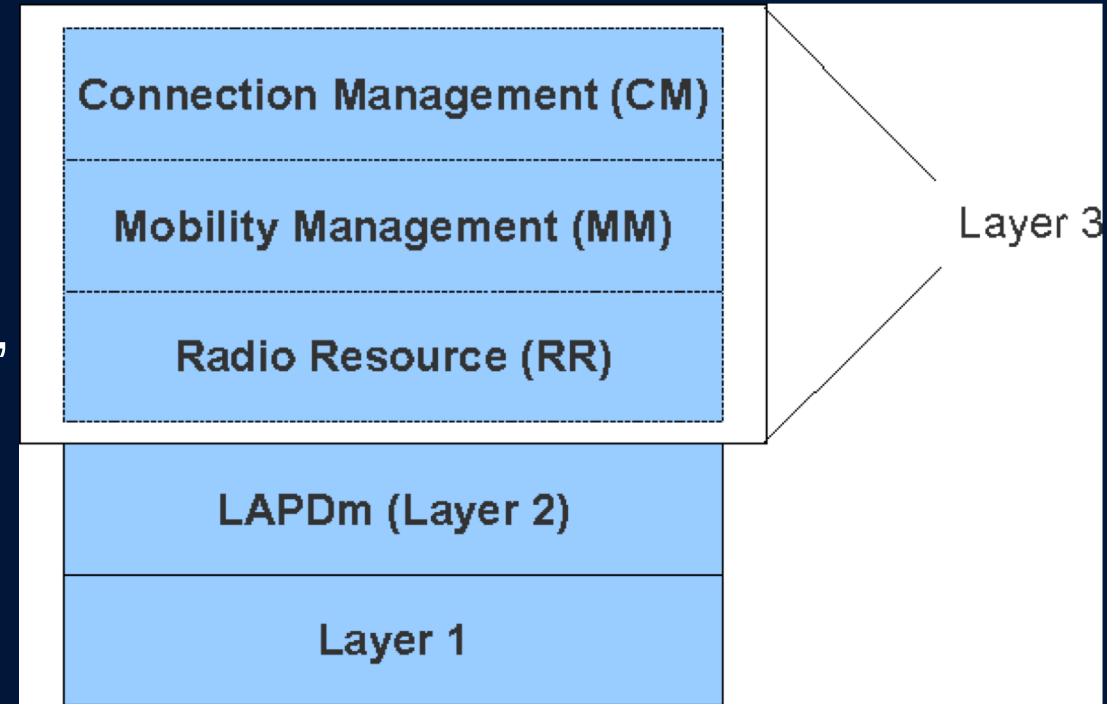
It runs «tasks»

You can find tasks responsible for some radio network layer, such as MM, SM, RR etcetera.

Once found the correct task, the task will dequeue a message usually and process it. Between those messages there are the radio messages you find defined in the specifications and you can audit.

Source **Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks**

Ralf-Philipp Weinmann



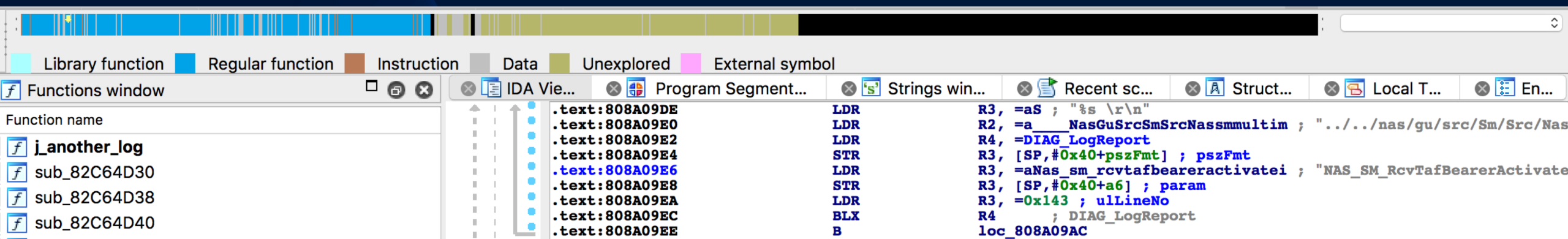
How to collect information

We have 3 main sources of information

1. The firmware
2. Runtime Information
3. Online Information

Let's check them out.

The Firmware



The screenshot shows the IDA Pro interface. At the top, there is a legend for function types: Library function (light blue), Regular function (dark blue), Instruction (orange), Data (grey), Unexplored (green), and External symbol (pink). Below the legend is the 'Functions window' showing a list of functions: j_another_log, sub_82C64D30, sub_82C64D38, and sub_82C64D40. The main window displays assembly code for the selected function. The code is as follows:

```
.text:808A09DE LDR R3, =aS ; "%s \r\n"
.text:808A09E0 LDR R2, =a____NasGuSrcSmSrcNassmmultim ; "../../../../nas/gu/src/Sm/Src/Nas
.text:808A09E2 LDR R4, =DIAG_LogReport
.text:808A09E4 STR R3, [SP,#0x40+pszFmt] ; pszFmt
.text:808A09E6 LDR R3, =aNas_sm_rcvtafbeareractivatei ; "NAS_SM_RcvTafBearerActivate
.text:808A09E8 STR R3, [SP,#0x40+a6] ; param
.text:808A09EA LDR R3, =0x143 ; ulLineNo
.text:808A09EC BLX R4 ; DIAG_LogReport
.text:808A09EE B loc_808A09AC
```

The first thing we need to do is to get our hands on the firmware and Reverse it to find exploitable memory corruptions over the air.

We will focus on the Huawei firmware.

We can find the file «sec_balong_modem.bin» in the smartphone filesystem.

The Android Kernel loads it, then it's passed to TEE (Trusted Execution Environment) for signature checks, then loaded into the baseband memory.

We can easily identify the code, and after some adjustments we can start to Reverse Engineer it.

Runtime Information

Runtime info are very helpful in the RE process and debugging

A previous talk mention «cshell», a runtime shell on the baseband

Sadly now it is disabled on newer versions.

However we found out that:

- When the baseband crashes, it will output back some errors to the Application Processor (Android) and log it on the filesystem.
- It is possible to read the baseband memory from the Android kernel, dumping the physical memory from 0x80400000

This helped us a lot to adjust our exploit.

Online Information

Obviously the Online specifications from 3GPP are mandatory to understand the systems.

We found on GitHub a old leaked version of part of this Baseband source code.

This was very helpful in the reverse engineering process.

There is also a existing project on the NVRAM format: <https://github.com/forth32/balong-nvtool>

Huawei Baseband Vulnerability Case Study



Preface

In this section we will show you the bug we used at Mobile pwn2own 2017 to exploit over the air a Huawei device baseband, gaining remote access on it.

First we will show you the bug

Then how we triggered it

Then how we exploited it

The Vulnerability

Our mobile pwn2own 2017 vulnerability was in the CDMA part of the baseband.

In detail, it was in the CDMA 1x SMS Transport Layer Message, in a function responsible for PRL messages.

A simplified version of the bug could be:

But what the heck is `memcpy_s`?

```
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;
}
```

memcpy_s

memcpy_s is a «secure» memcpy.

It takes 4 parameters instead of 3, source and destination size.

```
memcpy_s(void* destination, size_t dest_size, void* source, size_t src_size);
```

It checks that the copy doesn't exceed the destination buffer size, or the source buffer size.

Kills lot of bugs actually, purely by chance.

Our bug is not affected by memcpy_s because we control the offset of the copy.

The Vulnerability

In this message handler the message is parsed, and some offsets/lens are extracted. They are then added to a `byte_pos` without checks, this can lead to writing out of bound in the buffer (which is on the stack) leading to a exploitable stack overflow.

```
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;
}
```

Triggering and Delivering The Exploit



Setting up a CDMA network

Unfortunately there is no public open source software that allows you to run a proper CDMA 1x network with a SDR!

- There are many projects to setup cellular networks..
 - OpenBTS
 - OpenLTE
 - OpenAirInterface
 - ...
- OpenBTS with testcall to send arbitrary payload

None of them seems to support CDMA 1x

Do we have to build a new OpenCDMA?

- We came up with a different solution.

Setting up a CDMA network

We use the CMU200 together with a faraday cage, in order to gain better stability, avoid electromagnetic interference, and avoid to disturb other communications.

From the UI of the CMU200 we can adjust the network parameters (such as MCC, MNC).

After connecting a mobile phone to the CMU200 we can initiate phone calls, and send text messages from the UI.



Hacking into the Machine

Base unit

- Mainboard
- HDD with windows 3.x installed

Different optional link handler boards plugged in mainboard:

- B21: GSM/GPRS signaling hardware
- B83: CDMA2000 1xRTT signaling unit



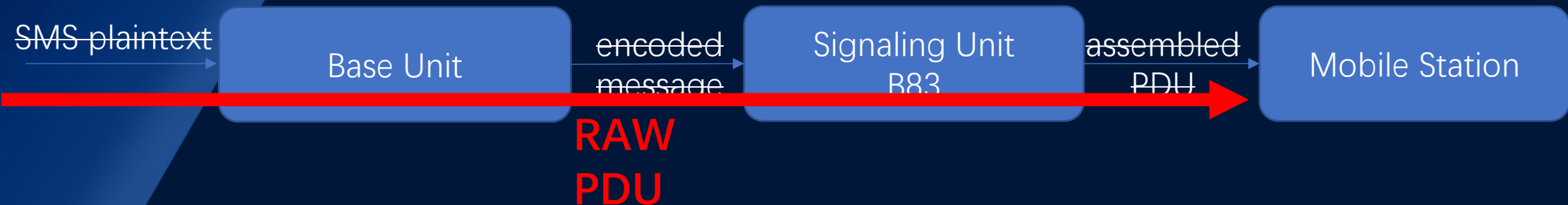
Hacking into the Machine

Base unit

- Mainboard
- HDD with windows 3.x installed

Different optional link handler boards plugged in mainboard:

- B21: GSM/GPRS signaling hardware
- B83: CDMA2000 1xRTT signaling unit



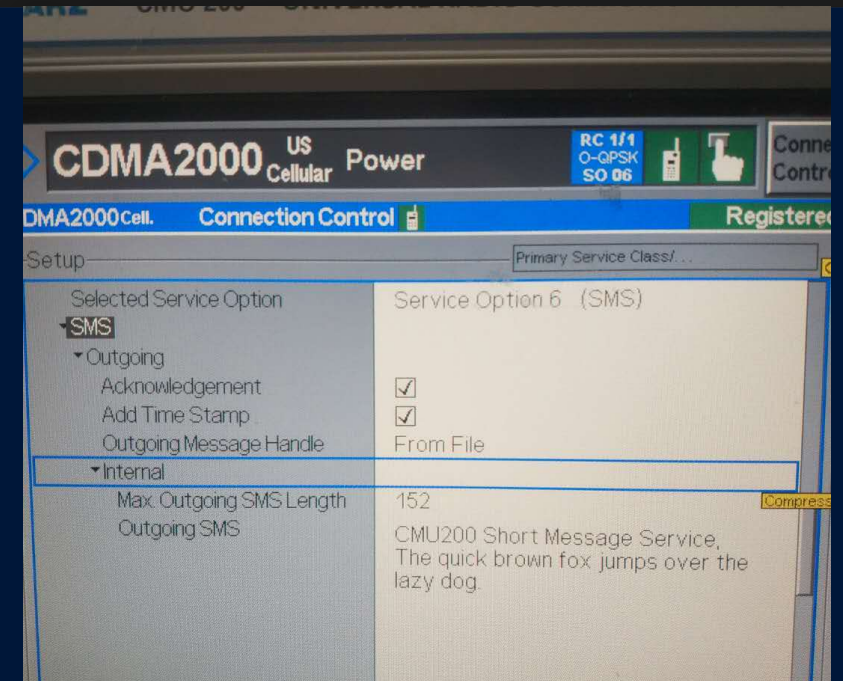
CMU200 Base Unit Reversing

CMU200 base unit is built on top of MS-DOS and Windows 3.x

Users can send a SMS from UI, or chose a predefined one to send.

It is all about PE reversing. Easy to locate the procedure that sending message to signaling unit.

```
total 144
drwxr-xr-x  13 501  20  416  7 29 16:54 .
drwx-----+ 95 501  20 3040  7 29 16:49 ..
-rwxr-xr-x@  1 501  20  459  7 29 16:37 AUTOEXEC.BAT
drwxr-xr-x@  20 501  20  640  7 29 16:37 CDMA2K
drwxr-xr-x  15 501  20  480  7 29 16:37 CMU
drwxr-xr-x  23 501  20  736  7 29 16:37 CMUDRV
-rwxr-xr-x   1 501  20 54645  7 29 16:37 COMMAND.COM
-rwxr-xr-x   1 501  20  420  7 29 16:37 CONFIG.SYS
drwxr-xr-x 100 501  20  3200  7 29 16:52 DOS
-rwxr-xr-x   1 501  20    0  7 29 16:37 NBOAFWU.VLA
-rwxr-xr-x   1 501  20   43  7 29 16:37 NOFWUPD.TXT
-rwxr-xr-x   1 501  20  462  7 29 16:37 SCANDISK.LOG
```



CMU200 Base Unit Patching

- `C2KMS.DL3` read content from internal file, and send it to B83 signaling unit as *link handler message*
- We can specify the teleservice identifier?
 - But cannot control full Bearer Data,
 - Let's go further
- Skip payload length check
- Force the signaling unit to upgrade

```
1. sh (sh)
sh-3.2$ cat INTERNAL/USERDATA/CDMA2K/OUT_MSG1.SMS
encoding=0
teleserviceid=4098
data=52265320434D552053686F7274204D65737361676520536572766963652054657374202D2042696E61727920534D
sh-3.2$
```

Table 3.4.2.1-1. SMS Point-to-Point Message Parameters

Parameter	Reference	Type
Teleservice Identifier	3.4.3.1	Mandatory
Service Category	3.4.3.2	Optional
Originating Address	3.4.3.3	Mandatory (1)
Originating Subaddress	3.4.3.4	Optional (1)
Destination Address	3.4.3.3	Mandatory (2)
Destination Subaddress	3.4.3.4	Optional (2)
Bearer Reply Option	3.4.3.5	Optional
Bearer Data	3.4.3.7	Optional

```
81 }
82 }
83 else if ( (_WORD)v29 == *(_WORD *)(*_DWORD *)(*_DWORD *) (a1 + 545) + 4) + 558 )
84 {
85     v2 = *(_DWORD *)(*_DWORD *) (a1 + 545) + 4);
86     LOWORD(v2) = *(_WORD *) (v2 + 564);
87     v28 = v2;
88     if ( !(_WORD)v33 )
89     {
90         if ( (signed __int16)v27 > 160 ) // length check for SMS
91             v33 = 14;
92         else
93             v31 = v27;
94     }
95 }
96 }
97 else if ( (_WORD)v29 == *(_WORD *)(*_DWORD *)(*_DWORD *) (a1 + 545) + 4) + 560 )
98 {
99     v3 = *(_DWORD *)(*_DWORD *) (a1 + 545) + 4);
100     LOWORD(v3) = *(_WORD *) (v3 + 566);
```

Signaling unit firmware format

Found upgrade functionality of the B83 unit.

- Upgrade occurs when self-check fails.
- Found firmware package YETIFLASH.FW&SASFLASH.FW
- Recover the format

MagicNum 0x53514646	CRC32 (4 bytes)	BlockSize (4 bytes)	Version (16 bytes)	Blob1Addr (4 bytes)	Blob1Size (4Bytes)	Blob1Data	Blob2Addr (4Bytes)	Blob2Size (4Bytes)	Blob2Data
------------------------	--------------------	------------------------	-----------------------	------------------------	-----------------------	-----------	-----------------------	-----------------------	-----------

- It is based on VxWorks for PowerPC!

```
2. sh (sh)
sh-3.2$ binwalk 0x0-0x900000
-----
DECIMAL      HEXADECIMAL  DESCRIPTION
-----
268          0x10C       Copyright string: "Copyright 1984-2004 Wind River Systems, Inc.8@"
15000        0x3A98      VxWorks operating system version "5.5.1" , compiled: "Oct 20 2010, 15:53:04"
15073        0x3AE1      Zlib compressed data, default compression
1048844      0x10010C    Copyright string: "Copyright 1984-2004 Wind River Systems, Inc.8@"
1063092      0x1038B4    VxWorks operating system version "5.5.1" , compiled: "Oct 20 2010, 16:03:00"
1064501      0x103E35    Zlib compressed data, default compression
```


Singling unit reversing

We are quite lucky this time:

- PowerPC decompiler
- symbol/name tables
- No signature check, only validating checksum

```
while ( v8 < 4 * MEMORY[0xF0040008] );  
}  
if ( MEMORY[0xF0040004] != v6 )  
{  
    printf("<FLASH> ", a2, a3, a4, a5, a6, &unk_620000);  
    v7 = printf("Bad checksum! Expected: 0x%08X, computed: 0x%08X\n", MEMORY[0xF0040004], v6);  
    goto LABEL_8;  
}  
return 0;  
}
```

594300	08080808	08080808	08080808	08080808	08080808	
594310	08080808	08080808	08080808	08080808	08080808	
594320	08080808	08080808	08080808	08080808	08080808	
594330	08080808	7A65726F	696E5F61	64647200		zeroin_addr
594340	7A65726F	4D657373	61676546	69656C64		zeroMessageField
594350	735F5F46	52313553	41535F4C	335F5052		s__FR15SAS_L3_PR
594360	4D5F5459	50450000	7A65726F	44736368		M_TYPE zeroDsch
594370	4F72646D	4D73675F	5F465232	31534153		OrdMMsg__FR21SAS
594380	5F4C335F	44534348	5F4F5244	4D5F5459		_L3_DSCH_ORDM_TY
594390	50450000	7A65726F	43736368	4F72646D		PE zeroCschOrdM
5943A0	4D73675F	5F465232	31534153	5F4C335F		Msg__FR21SAS_L3_
5943B0	43534348	5F4F5244	4D5F5459	50450000		CSCH_ORDM_TYPE
5943C0	79797661	6C000000	79797374	61727400		yyval yystart
5943D0	79797232	00000000	79797231	00000000		yyr2 yyr1
5943E0	79797067	6F000000	79797061	72736500		yypgo yyparse
5943F0	79797061	63740000	79796E65	72727300		yypact yynerrs
594400	79796C76	616C0000	79796578	63610000		yyval yyexca
594410	79796572	72666C61	67000000	79796465		yyerrflag yyde
594420	66000000	79796465	62756700	79796368		f yydebug yych
594430	6B000000	79796368	61720000	79796163		k yychar yyac
594440	74000000	78737075	746E5F5F	39737472		t xspu__9str
594450	65616D62	75665043	63690000	78737075		eambufPCci xspu
594460	746E5F5F	38737464	696F6275	66504363		tn__8stdiobufPCc
594470	69000000	78737075	746E5F5F	3766696C		i xspu__7fil
594480	65627566	50436369	00000000	78737075		ebufPCci xspu
594490	746E5F5F	3131696E	64697265	63746275		tn__11indirectbu
5944A0	66504363	69000000	78736765	746E5F5F		fPCci xsgetn__
5944B0	39737472	65616D62	75665063	69000000		9streambufPci
5944C0	78736765	746E5F5F	3766696C	65627566		xsgetn__7filebuf
5944D0	50636900	78736765	746E5F5F	3131696E		Pci xsgetn__11in
5944E0	64697265	63746275	66506369	00000000		directbufPci
5944F0	78736574	666C6167	735F5F39	73747265		xsetflags__9stre
594500	616D6275	66696900	78736574	666C6167		ambufii xsetflag
594510	735F5F39	73747265	616D6275	66690000		s__9streambufi
594520	78707574	5F636861	725F5F39	73747265		xput_char__9stre
594530	616D6275	66630000	78666C61	67735F5F		ambufc xflags__
594540	39737472	65616D62	75666900	78666C61		9streambufi xfla
594550	67735F5F	39737472	65616D62	75660000		gs__9streambuf
594560	7864726D	656D5F63	72656174	65000000		xdrmem_create

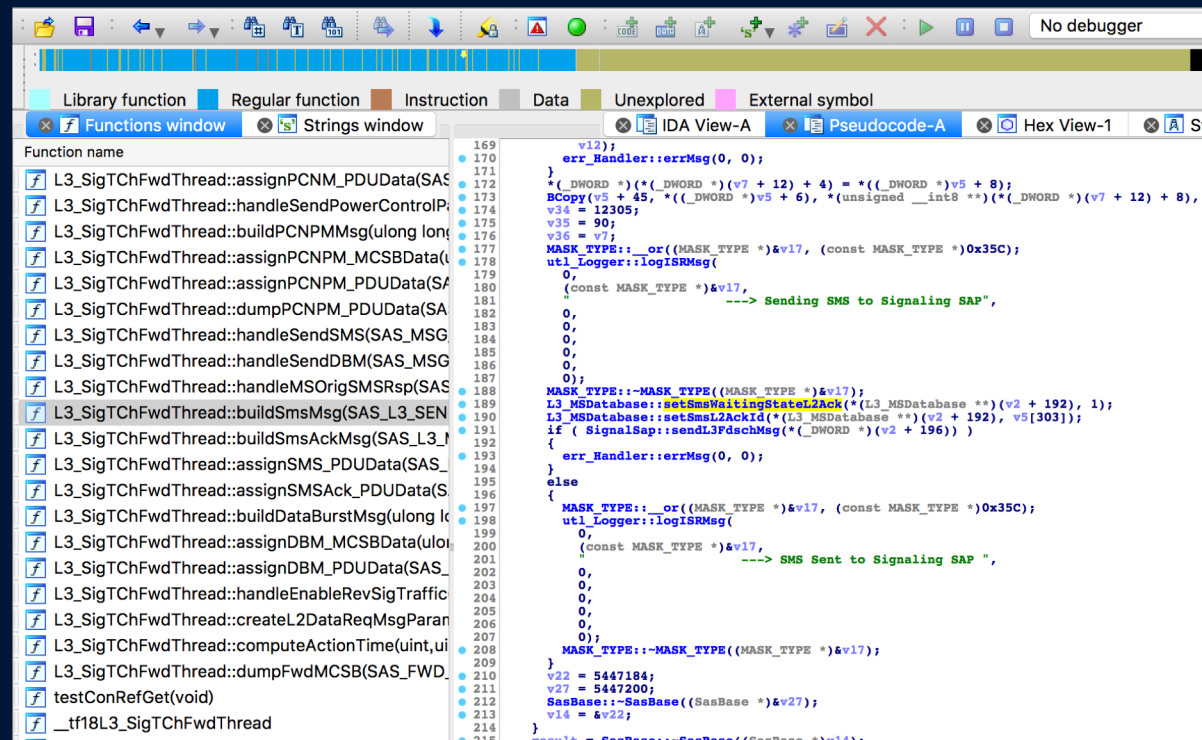
Signaling unit patching and repacking

Locate function `buildSmsMsg`, etc.

Patch it to carry arbitrary bearer data

Repack the firmware, and upgrade the B83 signaling unit!

With our own testcall, we are able to crash the baseband



The screenshot shows the IDA Pro interface with the 'Pseudocode-A' view selected. The function being viewed is `L3_SigTChFwdThread::buildSmsMsg(SAS_L3_SEN...)` starting at address 189. The pseudocode includes several operations: setting error handlers, copying data from memory, logging the start of an SMS message, and sending the message to the signaling SAP. A comment indicates the message is sent to the signaling SAP. The function ends with a return statement.

```
189 v12);
190 err_Handler::errMsg(0, 0);
191 }
192 *(_DWORD *)((_DWORD *) (v7 + 12) + 4) = *((_DWORD *)v5 + 8);
193 BCopy(v5 + 45, *((_DWORD *)v5 + 6), *((unsigned __int8 *) (_DWORD *) (v7 + 12) + 8), 0);
194 v34 = 12305;
195 v35 = 90;
196 v36 = v7;
197 MASK_TYPE: = _or((MASK_TYPE *)&v17, (const MASK_TYPE *)0x35C);
198 utl_Logger::logISRMsg(
199     0,
200     (const MASK_TYPE *)&v17,
201     "----> Sending SMS to Signaling SAP",
202     0,
203     0,
204     0,
205     0,
206     0);
207 MASK_TYPE: = -MASK_TYPE((MASK_TYPE *)&v17);
208 L3_MSDatabase::setSmsWaitingStateL2ACK(* (L3_MSDatabase **) (v2 + 192), 1);
209 L3_MSDatabase::setSmsL2AckId(* (L3_MSDatabase **) (v2 + 192), v5[303]);
210 if (SignalSap::sendL3FdschMsg(* (_DWORD *) (v2 + 196)) )
211 {
212     err_Handler::errMsg(0, 0);
213 }
214 else
215 {
216     MASK_TYPE: = _or((MASK_TYPE *)&v17, (const MASK_TYPE *)0x35C);
217     utl_Logger::logISRMsg(
218         0,
219         (const MASK_TYPE *)&v17,
220         "----> SMS Sent to Signaling SAP ",
221         0,
222         0,
223         0,
224         0,
225         0);
226     MASK_TYPE: = -MASK_TYPE((MASK_TYPE *)&v17);
227 }
228 v22 = 5447184;
229 v27 = 5447200;
230 SasBase: = -SasBase((SasBase *)&v27);
231 v14 = &v22;
232 }
233 result = SasBase: = SasBase((SasBase *)v14);
```

Exploitation



Reaching the vulnerable function

The exploit payload should be a malformed CDMA 1x SMS Transport Layer Message

Its SMS_MSG_TYPE field must be 00000000, indicating an SMS Point-to-Point message

Table 3.4-1. SMS Transport Layer Messages

Message Type	base station -> mobile station	mobile station -> base station	SMS_MSG_TYPE
SMS Point-to-Point	X	X	'00000000'
SMS Broadcast	X		'00000001'
SMS Acknowledge	X	X	'00000010'

All other values are reserved.

Reaching the vulnerable function

The message consists of TLV format PARAMETERS which must be set up properly to reach the vulnerable function

- Teleservice Identifier (PARAMETER_ID 00000000)
- Originating Address (PARAMETER_ID 00000010)

Table 3.4.2.1-1. SMS Point-to-Point Message Parameters

Parameter	Reference	Type
Teleservice Identifier	3.4.3.1	Mandatory
Service Category	3.4.3.2	Optional
Originating Address	3.4.3.3	Mandatory (1)
Originating Subaddress	3.4.3.4	Optional (1)
Destination Address	3.4.3.3	Mandatory (2)
Destination Subaddress	3.4.3.4	Optional (2)
Bearer Reply Option	3.4.3.5	Optional
Bearer Data	3.4.3.7	Optional

(1) For mobile-terminated messages (not present in mobile-originated messages)

(2) For mobile-originated messages (not present in mobile-terminated messages)

Reaching the vulnerable memcpy

The Bearer Data (PARAMETER_ID 00001000) is parsed in the vulnerable function

Table 3.4.2.1-1. SMS Point-to-Point Message Parameters

Parameter	Reference	Type
Teleservice Identifier	3.4.3.1	Mandatory
Service Category	3.4.3.2	Optional
Originating Address	3.4.3.3	Mandatory (1)
Originating Subaddress	3.4.3.4	Optional (1)
Destination Address	3.4.3.3	Mandatory (2)
Destination Subaddress	3.4.3.4	Optional (2)
Bearer Reply Option	3.4.3.5	Optional
Bearer Data	3.4.3.7	Optional

(1) For mobile-terminated messages (not present in mobile-originated messages)

(2) For mobile-originated messages (not present in mobile-terminated messages)

Reaching the vulnerable memcpy

The Bearer Data (PARAMETER_ID 00001000) is parsed in the vulnerable function

- Which in turn consists of TLV format SUBPARAMETERS
- It should indicate itself a PRL message through properly set SUBPARAMETERS

One or more occurrences of the following subparameter record:

SUBPARAMETER_ID	8
SUBPARAM_LEN	8
Subparameter Data	8× SUBPARAM_LEN

Reaching the vulnerable memcpy

Message Display Mode (SUBPARAMETER_ID 00001111)

- MSG_DISPLAY_MODE field must be 0x03
- RESERVED field must be 0x10

Field	Length (bits)
SUBPARAMETER_ID	8
SUBPARAM_LEN	8
MSG_DISPLAY_MODE	2
RESERVED	6

Table 4.5-1. Bearer Data Subparameter Identifiers

Subparameter	SUBPARAMETER_ID Value
Message Identifier	'00000000'
User Data	'00000001'
User Response Code	'00000010'
Message Center Time Stamp	'00000011'
Validity Period - Absolute	'00000100'
Validity Period - Relative	'00000101'
Deferred Delivery Time - Absolute	'00000110'
Deferred Delivery Time - Relative	'00000111'
Priority Indicator	'00001000'
Privacy Indicator	'00001001'
Reply Option	'00001010'
Number of Messages	'00001011'
Alert on Message Delivery	'00001100'
Language Indicator	'00001101'
Call-Back Number	'00001110'
Message Display Mode	'00001111'
Multiple Encoding User Data	'00010000'
Message Deposit Index	'00010001'
Service Category Program Data	'00010010'
Service Category Program Results	'00010011'

The vulnerable code

The vulnerable code basically sorts the sub parameters in the Bearer Data

And the buffer overflow happens right here

It seems trivia for the rest part of the exploitation

- Exploit a classical stack overflow just like in the 90s
 - No NX/ASLR/Stack Canary

Is it true?

Exploit a stack overflow(?) in baseband

Multiple paths to the vulnerable function exist

- Two of them are not reachable over the air (only used in MO Message)
- One of them are reachable through MT Message, but the buffer is inside global variable section, rather than on stack

Who said it is a stack overflow?

Exploit a stack overflow(?) in baseband

There do exist one path to the vulnerable function with the buffer on stack

- However, it is only used when reading out an SMS from USIM

Looks we are out of options

Where there is a will, there is a way

We discovered a deep but stable path all the way down to the vulnerable function, following this seemingly useless path

Exploit a stack overflow(!) in baseband

The whole process of handling a PRL message goes like this

- Receive the message over the air
- Decode the message (1st time with the vulnerable function but buffer not on stack)
- Encode the message
- Write the message into USIM
- Read the message out from USIM
- Decode the message (2nd time with the vulnerable function and buffer on stack)

Construct the payload

The payload must survive the first decoding & encoding cycle, and overflow the stack in the second decoding process

Not so trivial right?

Construct the payload

Let's do some simple math

The payload: \mathbf{x}

Decoding function – $dec(\mathbf{x})$

Encoding function – $enc(\mathbf{x})$

Stack overflow ROP chain: \mathbf{p}

Goal: Find an \mathbf{x} for a given \mathbf{p} such that $\mathbf{p} = dec(enc(dec(\mathbf{x})))$

Construct the payload

Solve a mathematical problem in a more mathematical way

Our goal:

- Find an \mathbf{x} for a given \mathbf{p} such that $\mathbf{p} = \text{dec}(\mathbf{x})$ and $\mathbf{x} = \text{enc}(\text{dec}(\mathbf{x}))$
- \mathbf{x} is the *fixed point* of function $\text{enc}(\text{dec}(\mathbf{x}))$

So that $\mathbf{p} = \text{dec}(\mathbf{x}) = \text{dec}(\text{enc}(\text{dec}(\mathbf{x})))$

Such a payload works for arbitrary levels of decoding & encoding cycle

Furthermore, the CMU200 machine restricts the length of TP layer message to be less than 130 bytes

Construct the payload

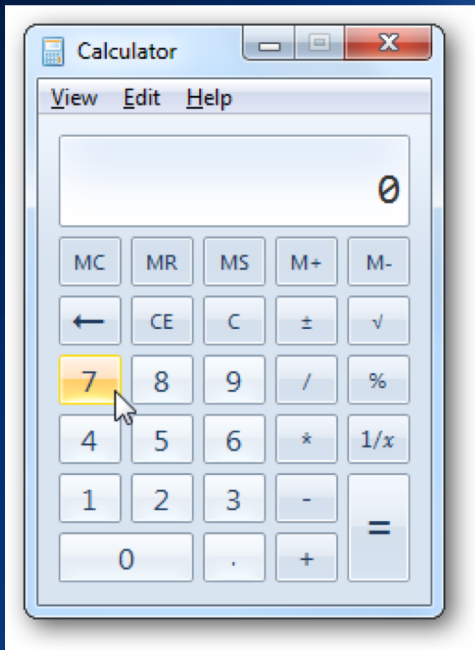
Constructing such a payload is not trivia, but possible

For more details please refer to the white paper

And I adapted it to one CTF challenge

- Mighty Dragon (OCTF/TCTF 2018 Quals)
- Named after the codename of the modem “balong” (霸龙)

Payload Execution and Capabilities



So we gained RCE on the baseband, but how can we demonstrate it?

We cannot pop «calc.exe» like on windows, there is no clear UI.

We decided to change the phone IMEI to give a visual confirmation of successful exploit.

It can be viewed in the Settings of the Phone.

From the baseband we have access all calls/SMS/mobile internet traffic, sniffing it or tampering it.

DEMO



Vendor Response

Huawei quickly patched the issue releasing updates

Good communication with their security response team

We follow up after pwn2own with the vendor

Responsive feedback and they seems to care about security.

Escaping the Baseband and Further Compromise

Escaping the baseband was not required by the contest rules, so we didn't do it.

This is not a very publicly studied topic, but there are good examples of similar scenarios:

- https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html (using unrestricted DMA to overwrite AP memory)
- https://comsecuris.com/blog/posts/path_of_least_resistance/ (path traversal in a usermode component allows an attacker to modify files in the AP filesystem)

An attacker can chain a baseband escape to a RCE, just like a Sandbox escape can be chained to a Browser RCE

Gain complete control of the target device.

The attack surface is significant, since a lot of information must be exchanged between the baseband and the AP.

Conclusions

We demonstrated that a Baseband RCE is not only possible, but also practical for a determined attacker.

Basebands are very complex software, with a huge remote attack surface.

They are written in memory unsafe language (c/cpp mainly) and they lack of even basic mitigations.

It should not surprise that a determined attacker can gain RCE on them.

We hope in the future more mitigations are deployed in basebands, and hopefully in the long term a switch to more memory safe languages will happen.

This long timeframe should be addressed temporarily by more security scrutiny.

Acknowledgements

Wushi

Zhao

Anton

Wenkai Zhang

Haijiang Xie



Questions?

