TENCENT SECURITY CONFERENCE 2018
2018腾讯安全国际技术峰会

# A look at the Samsung Shannon Baseband.

@amatcama

## Amat Cama

- Independant Security Researcher / **Securin Technology**
- CTF player @ Shellphish.
- Exploitation and Reverse Engineering.
- Currently interested in Hypervisors and Baseband reseach.

## Securin Technology

- Security Research.
- Trainings (Vulnerability Research, RE, Exploitation).
- Consultation (Code Audits, Vulnerability Assessments).

## Agenda

- Prior Work.
- Cellular Networks ? Baseband ?
- The Shannon Baseband.
- Hunting For Bugs.
- Demo.
- “Advanced” Debugging.
- Conclusions.

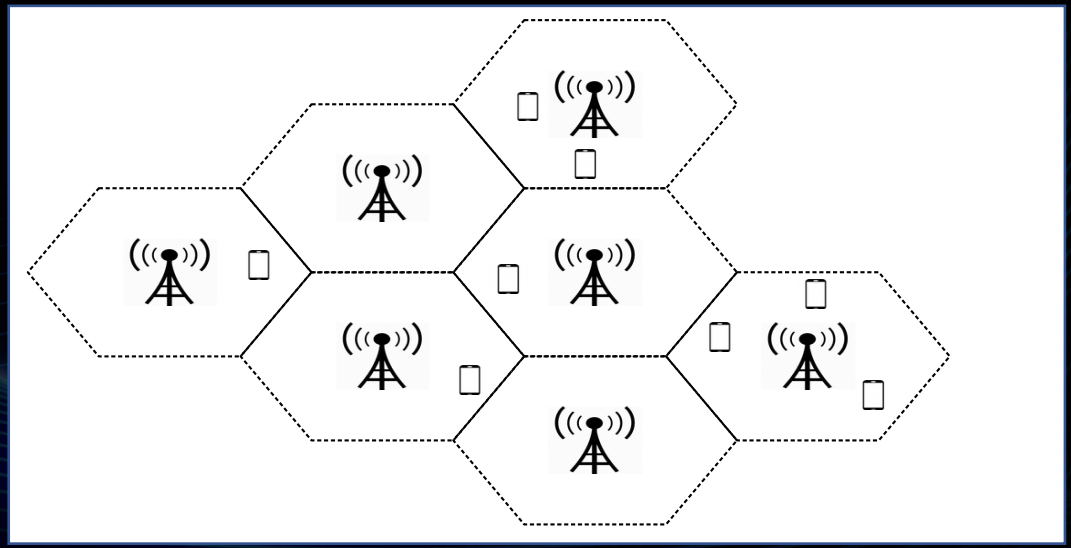
# Prior Work

- “**Breaking Band - reverse engineering and exploiting the shannon baseband**” – *Nico Golde and Daniel Komaromy.*
- “***Exploitation of a Modern Smartphone Baseband***” – *Marco Grassi, Muqing Liu and Tianyi Xie.*

# Cellular Networks ?
# Baseband ?

## What is a Cellular Network ?

- Mobile communication network.
- “Cells” are land areas covered by a *base transciever station* (***BTS***).
- To cover a large area, the cells are used in junction: *A Cellular Network*.
- Technically could be any kind of network, today mostly *Mobile Phone Network*.

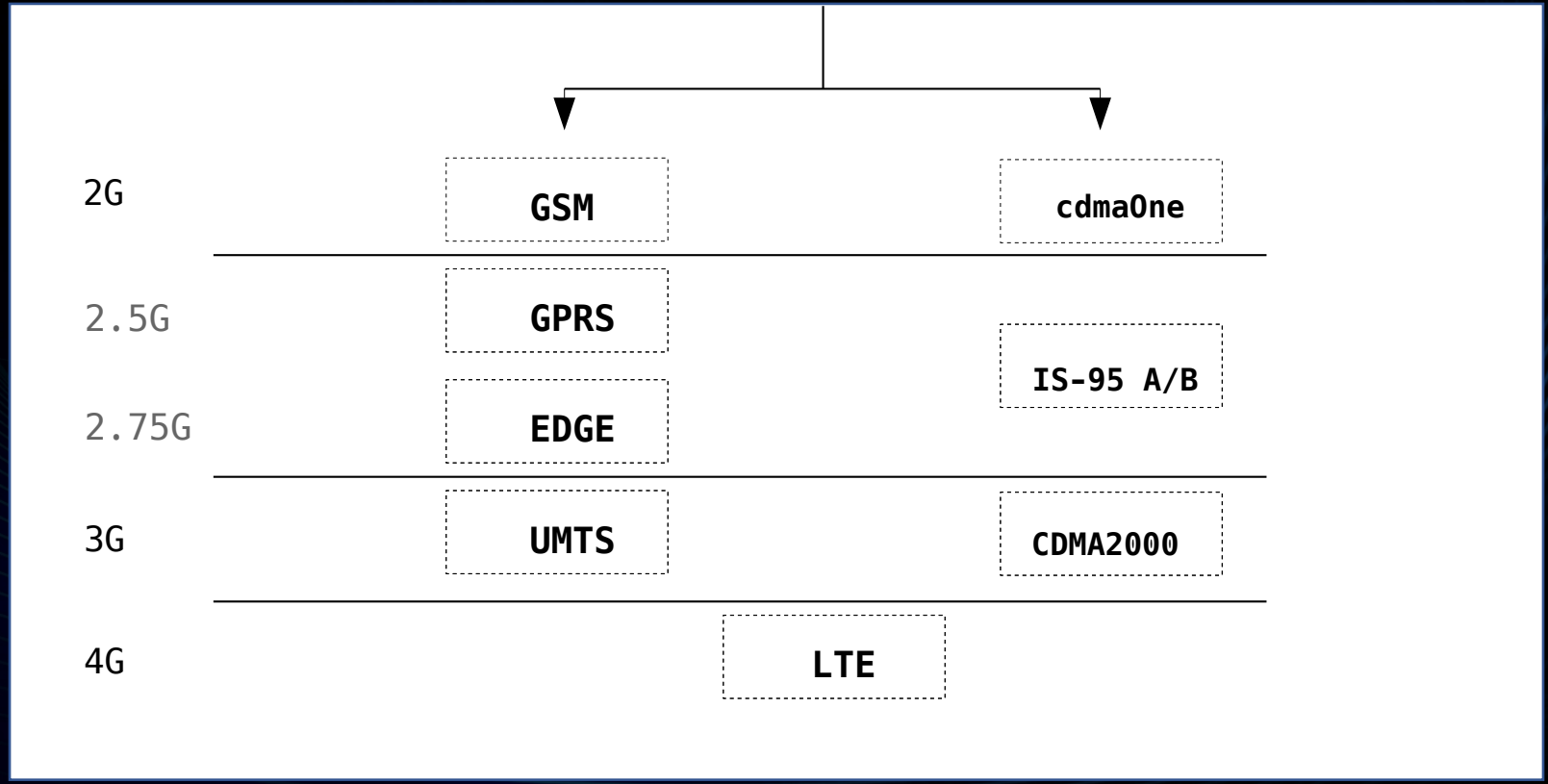


## The technologies and standards (I)

- A number of technologies and standards developped.
- Different generations with improving speeds and capacity.
- Competing technologies for different generations.

# The technologies and standards (II)

- Mainly two branches: ***GSM*** branch and ***CDMA*** branch



## The technologies and standards (III)

- *3GPP* is a collaboration agreement with a number of telecommunication standard bodies.
- Provides maintenance and development of the GSM *Technical Specifications (TS)*
  - GSM
  - GPRS / EDGE
  - UMTS
  - LTE
- Is Comprised of bodies such as the *European Telecommunications Standards Institute (ETSI)*.
- The technical standards provide detailed information on the structure of messages exchanged.

# The Protocol Stack

| | GSM | GPRS/EDGE | LTE |
|---|---|---|---|
| Network Layer | **CM** - Connection Management<br>**MM** - Mobility Management<br>**RR** - Radio Resource Management | **SM** - Session Management<br>**GMM** - GPRS Mobility Management<br>**GRR** - GPRS Radio Resource Management | **IP** - Internet Protocol<br>**NAS** - Non-Access Spectrum<br>**RRC** - Radio Resource Control |
| Data Link Layer | **LAPDm** | **SNDCP** - Subnetwork Dependent Convergence Protcol<br>**RLC/LLC** - Radio/Logical Link Control<br>**MAC** - Media Access Control | **PDCP** - Packet Data Convergence Protcol<br>**RLC/LLC** - Radio/Logical Link Control<br>**MAC** - Media Access Control |
| Physical Layer | | | |

## The Baseband (I)

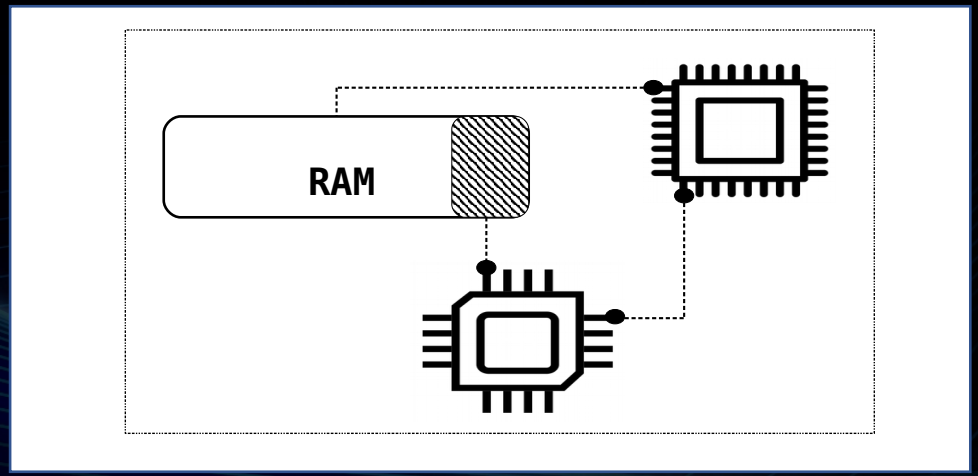
- Component of the phone in charge of handling communication with the mobile network.
- Deals with low level radio signal processing.
- Supports a number of standards (GSM, 3G, 4G, 5G, cdmaOne, CDMA2000, ...).
- Basically the main “interface” to the mobile network.

## The Baseband (II)

- A number of different implementations.
- Qualcomm owns most of the market.
- Qualcomm: Galaxy, iPhone, OnePlus, Pixel, Xperia, HTC, LG, ASUS, Motorola, ...
- Huawei: Mate 10, P20, Honor 9, ...
- **Samsung**: Galaxy S6, S7, S8, S9, ...
- Intel: iPhone Xs/r,iPhone X, iPhone 8, ...
- Mediatek: Xiaomi, Tecno, ...

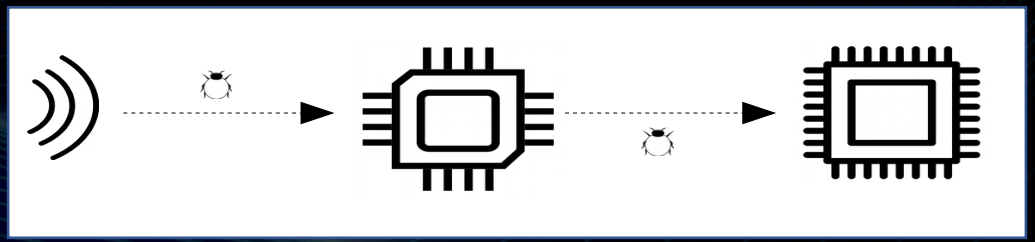
## The Baseband (III)

- he most common architecture today: baseband firmware runs on a dedicated chip; the *cellular processor (**CP**)*.
- This chip is tasked with all of the radio processing.
- The code is generally written in low level languages such as C/C++.
- A communication interface between **CP** and **AP** (Application Processor) such as shared memory, serial or interrupts.



## The Baseband (IV)

- Getting code execution on the *CP* doesn't necessarily result in owning the whole device.
- A number of attacks can be performed:
  - → Redirect/Intercept phone calls.
  - → Redirect/Intercept SMS.
  - → Modify Internet traffic.
  - → ...
- A step further; attack the *AP* through the IPC mechanisms and gain full control of the device.



# The Shannon Baseband

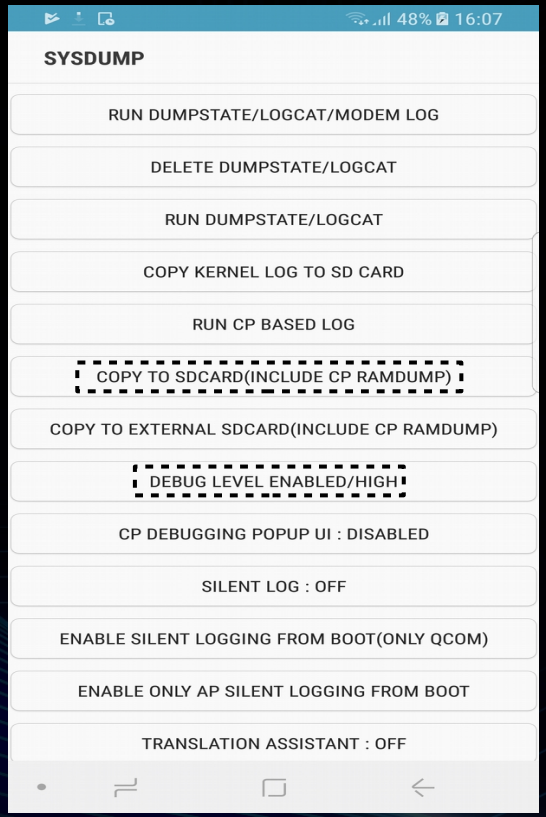
## About Shannon

- Samsung’s Baseband implementation.
- Typically ships with phones featuring the Exynos SoC.
- e.g: most non-US Galaxy phones.
- A RTOS running on an ARM Cortex R7.

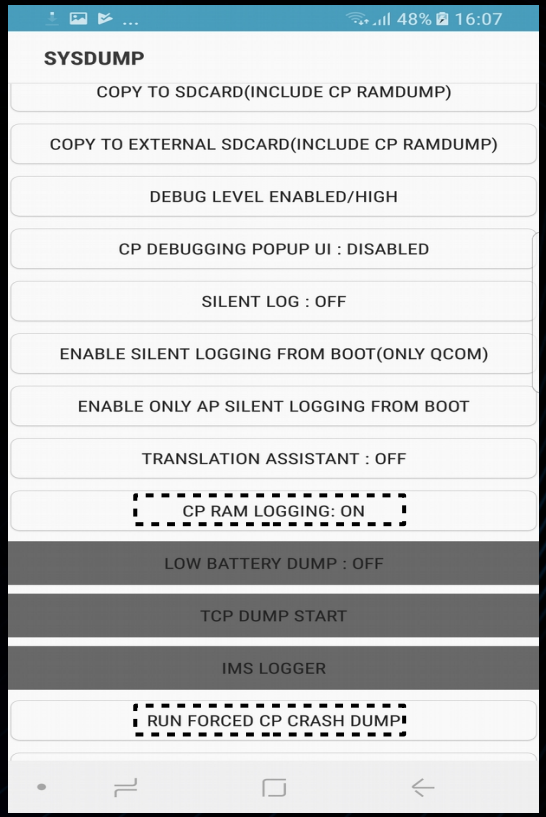
## Obtaining the code (I)

- The modem firmware can be obtained from the phone’s firmware images.
- However it is encrypted and doesn’t seem to be an easy way to decrypt it.
- Luckily it is possible to make the phone generate modem RAM dumps.
- Dialing the code *#9900# brings up the *SYSDUMP* menu.

# Obtaining the code (II)




Page 1

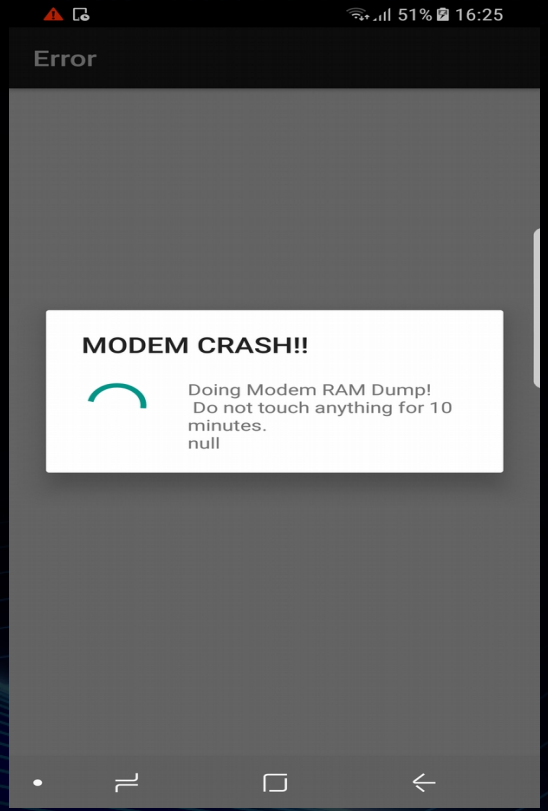



Page 2

## Obtaining the code (III)

- Tap on the `DEBUG LEVEL ENABLED/` option and set it to `High`. The phone will reboot.
- Reopen the SYSDUMP menu, scroll down and tap on the `CP RAM LOGGING` option and set it to `On`. The phone will reboot.
- Reopen the SYSDUMP menu and scroll all the way down, tap the `RUN FORCED CP CRASH DUMP` option. The phone will reboot and go into the ram upload mode. Hold the power and volume down button for 10 seconds to turn the phone off and then power it back on.
- Reopen the SYSMDUMP menu and tap the `COPY TO SDCARD(INCLUDE CP RAMDUMP)` option.
- Now in the folder `/sdcard/log` of the device, we have the log files including the ram dump. Largest file in the folder and has a name of the following format `cpcrash_dump_YYYYMMDD_HHSS.log`

# Obtaining the code (IV)




## Loading Code in IDA

- The CP Boot Daemon (/sbin/cbd) handles powering on the modem and processing RAM dumps amongst other things.
- Boot code can be found at the start of the encrypted modem image in the firmware packages.
- By reversing the cbd and boot, we can translate the file offsets of the RAM dump to virtual addresses:

  **0x40000000**

  **0x8000000**

  **0x4000000**

  **0x20000**

  **0x4800000**

  ...

## Identifying Tasks

- We need to identify the different tasks run by the RTOS.
- Start reversing from RESET Exception Vector Handler…
- Look at the start of the different memory regions and you recognize the Exception Vector Table in one of them.
- A linked list contains all the different tasks' entry points, corresponding stack frames and task names (very useful).
- Traverse the list and identify all the tasks.

## The Tasks (I)

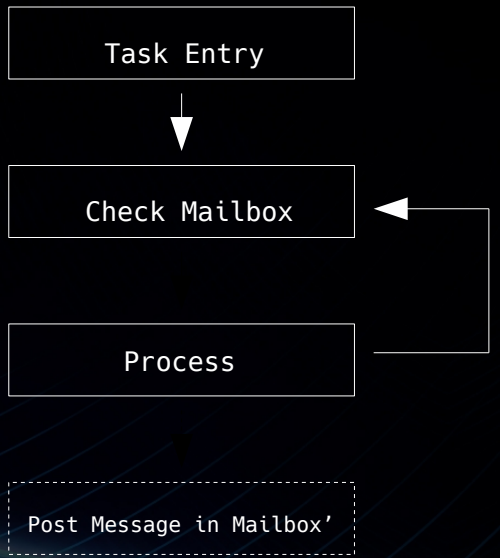
- We end up with a list of tasks with different names, some of them self-explanatory, some of them misleading, some of them hard to understand.
- **MM** (**M**obility **M**anagement ?)
- **SAEL3** (**S**ystem **A**rchitecture **E**volution / LTE **L**ayer **3** ?)
- SNDCP
- **CC** (**C**all **C**ontrol ?)
- **SM** (**S**ession **M**anagement ?)
- LLC
- ...

# The Tasks (II)

| | GSM | GPRS/EDGE | LTE |
|---|---|---|---|
| Network Layer | **CM** - Connection Management<br>**MM** - Mobility Management<br>**RR** - Radio Resource Management | **SM** - Session Management<br>**GMM** - GPRS Mobility Management<br>**GRR** - GPRS Radio Resource Management | **IP** - Internet Protocol<br>**NAS** - Non-Access Spectrum<br>**RRC** - Radio Resource Control |
| Data Link Layer | **LAPDm** | **SNDCP** - Subnetwork Dependent Convergence Protcol<br>**RLC/LLC** - Radio/Logical Link Control<br>**MAC** - Media Access Control | **PDCP** - Packet Data Convergence Protcol<br>**RLC/LLC** - Radio/Logical Link Control<br>**MAC** - Media Access Control |
| Physical Layer | | | |

## The Tasks (III)

- Different tasks are used for different components and layers of the protocol stacks.
- Tasks communicate with each other using a *mailbox* system.
- Tasks are pretty much *while* loops waiting to process messages (from other tasks).

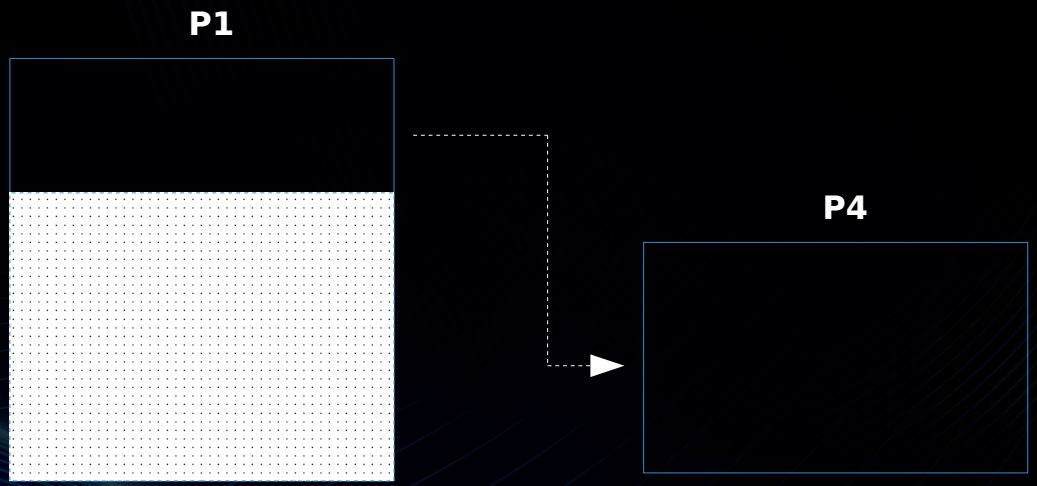


## The Tasks (IV)

- Pick a task and start reversing.
- The code is pretty generous in that it contains a **lot** of strings.

## The Heap (I)

- The heap is segmented into different “pools” numbered **1** through **6**.
- Most of the allocations happen from pool number **4** (*the main pool*).
- The *main pool* is a client of pool **1**, 0x700000 bytes is allocated for it.

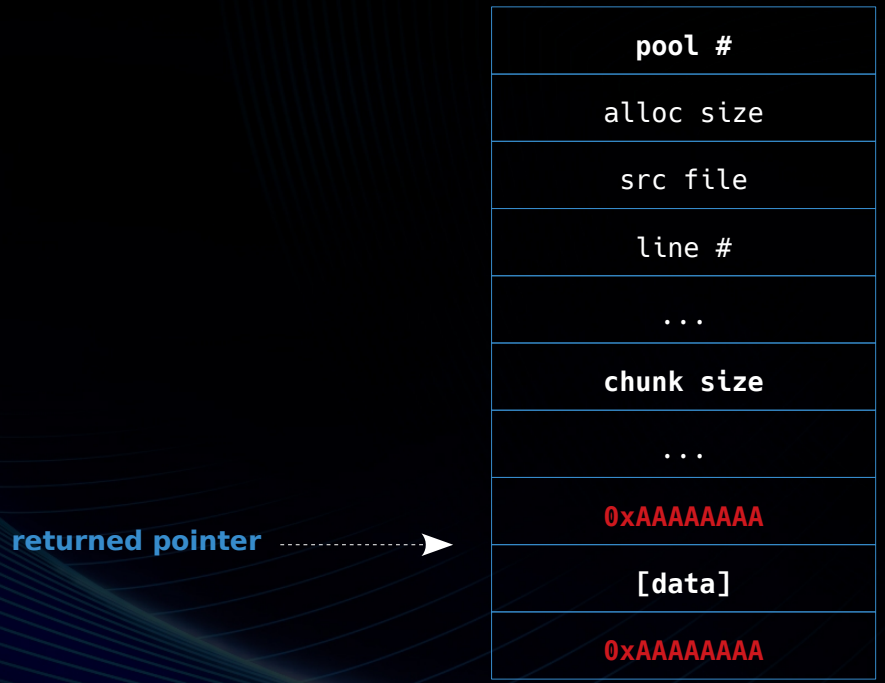


## The Heap (II)

- The allocation and de-allocation functions are:
  - **pal_MemAlloc**(int pool_no, size_t alloc_size, char * filename, int line_no)
  - **pal_MemFree**(void **ptr, char * filename, int line_no)

## The Heap (III)

- An allocated heap chunk from the main pool has a header of size **0x20** bytes:



## The Heap (IV)

- The pool is segmented into “pages” for different chunk sizes
- The 0x700000 bytes allocated for the main pool contain the pool header at the start:
  - A *control structure* describing free and used chunks (bitmap)
  - The *size* of the pool
  - The *start* and *end* address of the pool

# Hunting for Bugs

## Setting up an environment (I)

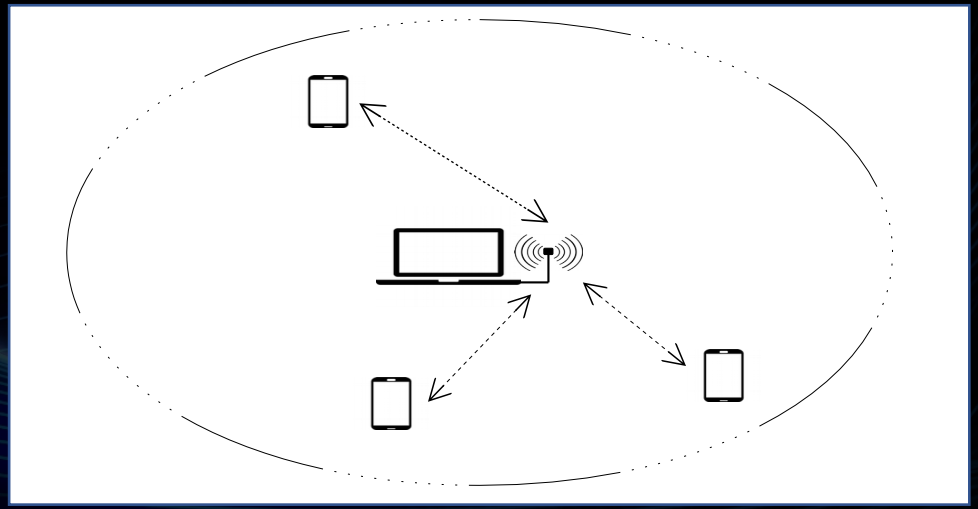
The goal is to be able to send arbitrary data the the baseband.

Need to operate our own cellular network.

Can be achieved with a *Software Defined Radio (**SDR**)*.

The Mobile Network Stack / Standard is implemented in software that runs on our computers.

The SDR (device) is a general purpose transciever that supports different frequencies.



## Setting up an environment (II)

A number of different options for the SDRs.

- *BladeRF x40*: $420.00
- ***BladeRF x115***: $650.00
- ***USRP B200***: $675.00
- ***LimeSDR***: $300.00
- *UmTRX*: $950.00 - $1300.00

## Setting up an environment (III)

A number of different options for software implementation of the standards.

***YateBTS:***

- Clean code, easy to modify.
- Supports *bladeRF*.
- GSM and GPRS.
- Easy to compile and run.

***OpenBTS*** *(OpenBTS-UMTS):*

- Clean code, easy to modify.
- Good support for *USRP* and *UmTRX*.
- GSM, GPRS, 3G.
- Easy to compile and run.

## Setting up an environment (IV)

*OpenBSC* (OsmoNITB, OsmBTS, …):

- Good support for *USRP*, *LimeSDR* and *UmTRX*.
- Compiling wasn't easy.
- Clean code, easy to modify.
- GSM + GPRS.

*OpenAirInterface:*

- Hard to compile and run.
- Good support for *USRP*.
- 4G/LTE.

***srsLTE:***

- Very easy to compile and run.
- 4G/LTE.
- Good support for *USRP* and *LimeSDR*.
- Clean code, easy to modify.

## Setting up an environment (V)

Provisioned or programmable *SIM Cards* because 3G and 4G do not support open authentication.

*Faraday Cage / RF Enclosure* because in most countries, operating a cell network without permission is **illegal**!

## Debugging The Phone

Everytime the modem crashes we get a RAM dump.

Luckily the dump contains the state of the registers at the time of the crash, therefore we have pretty decent post-mortem debugging capabilities.

Write a script to process the dumps and do useful stuff (registers, peeking at memory).

## Digging into the code (I)

Back to picking a task to have a closer look at.

Interesting area to look at is the Layer 3 / NAS:

- Layer 3 Messages are comprised of *Information Elements (IEs)*.
- IEs are *V*, *LV*, *TLV*.
- We can imagine code patterns that trust the length field without checks.

## Digging into the code (II)

Let’s clarify with a few examples.

The CC task stands for **C**all **C**ontrol.

Call control is a part of Connection Management in the GSM protocol stack and it sits at Layer 3.

There are a number of CC messages that can be sent by the network to the phone.

## Digging into the code (III)

There are a number of CC messages that can be sent by the network:

- Alerting
- Call proceeding
- Connect
- Connect acknowledge
- Disconnect
- Facility
- Notify
- **Progress**
- Setup
- **Status**
- ...

# Progress Message Bug

**Table 9.67/3GPP TS 24.008: PROGRESS message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Call control protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2 |
| | Progress message type | Message type 10.4 | M | V | 1 |
| | Progress indicator | Progress indicator 10.5.4.21 | M | LV | 3 |
| 7E | User-user | User-user 10.5.4.25 | O | TLV | 3-131 |

# Status Message Bug

**Table 9.74/3GPP TS 24.008: STATUS message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Call control protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2 |
| | Status message type | Message type 10.4 | M | V | 1 |
| | Cause | Cause 10.5.4.11 | M | LV | 3-31 |
| | Call state | Call state 10.5.4.6 | M | V | 1 |
| 24 | Auxiliary states | Auxiliary states 10.5.4.4 | O | TLV | 3 |

## Digging into the code (V)

These vulnerabilities are pretty trivial. Essentially doing this:

```
1  void vuln(unsigned __int8 *tlv)
2  {
3      unsigned __int8 tbuf[N];
4      unsigned __int8 len;
5
6      ....
7
8      len = tlv[1];
9      ....
10
11     memcpy(tbuf, &tlv[2], len)
12     ...
13 }
```

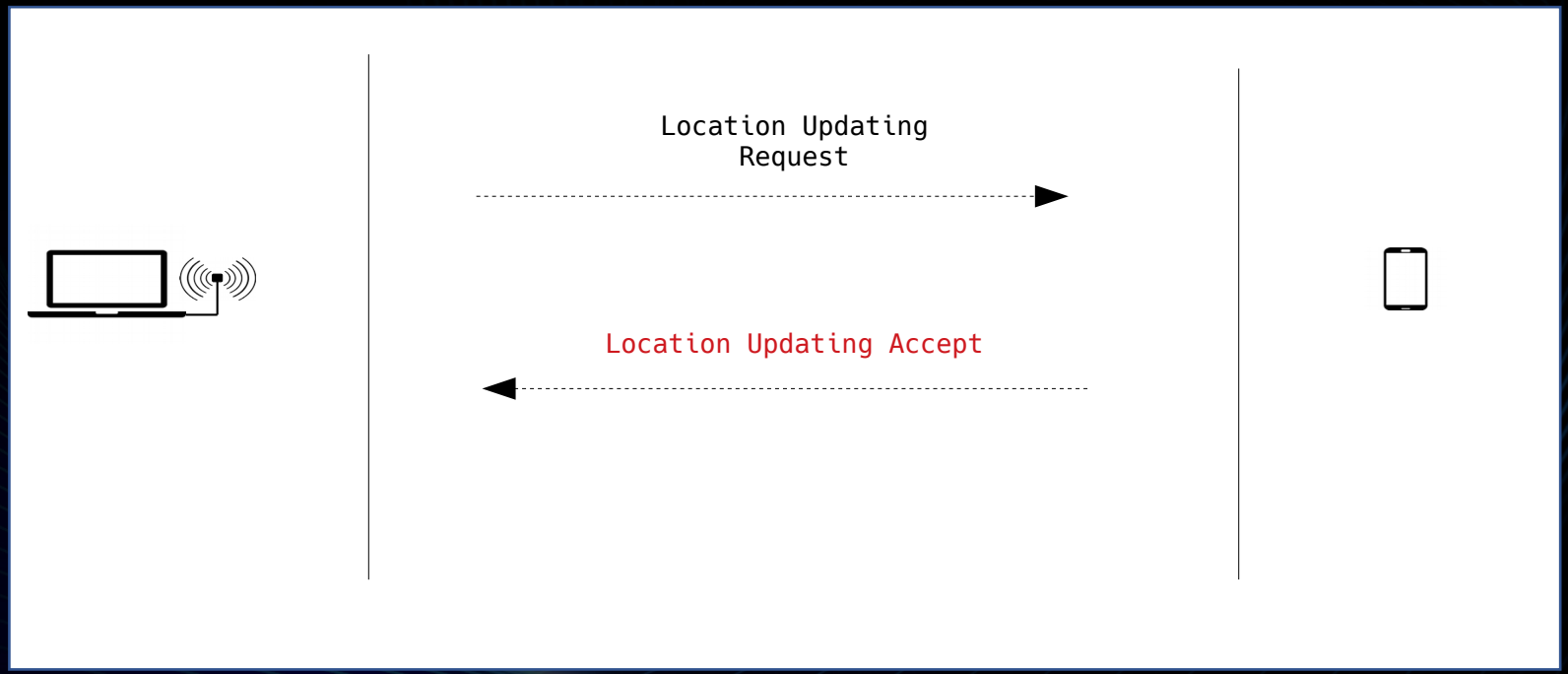
## Digging into the code (III)

Another interesting component is the Mobility Management component. Again, there are a number of messages that can be sent by the network:

- Authentication request
- **Location updating accept**
- Location updating reject
- MM information
- MM status
- Identity request
- ...

## Location Updating Accept Bug

When the phone first connects to the network:



# Location Updating Accept Bug

**Table 9.2.15/3GPP TS 24.008: LOCATION UPDATING ACCEPT message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Mobility management protocol discriminator | Protocol discriminator 10.2 | M | V | ½ |
| | Skip Indicator | Skip Indicator 10.3.1 | M | V | ½ |
| | Location Updating Accept message type | Message type 10.4 | M | V | 1 |
| | Location area identification | Location area identification 10.5.1.3 | M | V | 5 |
| 17 | Mobile identity | Mobile identity 10.5.1.4 | O | TLV | 3-10 |
| A1 | Follow on proceed | Follow on proceed 10.5.3.7 | O | T | 1 |
| A2 | CTS permission | CTS permission 10.5.3.10 | O | T | 1 |
| 4A | Equivalent PLMNs | PLMN list 10.5.1.13 | O | TLV | 5-47 |
| 34 | Emergency Number List | Emergency Number List 10.5.3.13 | O | TLV | 5-50 |
| 35 | Per MS T3212 | GPRS Timer 3 10.5.7.4a | O | TLV | 3 |

## Location Updating Accept Bug

The location updating accept message contains some information such as the *Mobile Identity* and *Emergency Number List*.

> “The purpose of the *Emergency Number List* information element is to encode emergency number(s) for use within the country where the IE is received”.
>
> It has a maximum length of **50** bytes.

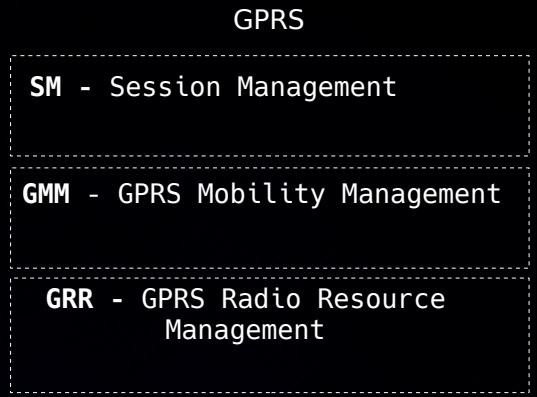
# Location Updating Accept Bug

```
1  int ParseEmergencyNumberList(unsigned __int8 *buf, unsigned int len, EmergencyNumber_t *output)
2  {
3    int idx;
4    unsigned int i;
5    int8_t emnumlen;
6
7    idx = 0;
8    for ( i = 0; i < len; i = i + emnumlen + 1) )
9    {
10     emnumlen = buf[i];
11     if ( !buf[i] )
12       break;
13     ParseEmergencyNum(buf[i], &buf[i + 1], &output[idx]);
14     idx = idx + 1;
15   }
16   return idx;
17 }
```

## The Mobile Pwn2Own Bug (I)

Decided to look at GPRS since it seems complicated?

Start by reading the standards and looking at the GPRS Session Management Messages.



# The Mobile Pwn2Own Bug (II)

The ACTIVATE PDP CONTEXT ACCEPT message looks good.

**Table 9.5.2/3GPP TS 24.008: ACTIVATE PDP CONTEXT ACCEPT message content**

| IEI | Information Element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2– 3/2 |
| | Activate PDP context accept message identity | Message type 10.4 | M | V | 1 |
| | Negotiated LLC SAPI | LLC service access point identifier 10.5.6.9 | M | V | 1 |
| | Negotiated QoS | Quality of service 10.5.6.5 | M | LV | 13-21 |
| | Radio priority | Radio priority 10.5.7.2 | M | V | 1/2 |
| | Spare half octet | Spare half octet 10.5.1.8 | M | V | 1/2 |
| 2B | PDP address | Packet data protocol address 10.5.6.4 | O | TLV | 4-24 |
| 27 | Protocol configuration options | Protocol configuration options 10.5.6.3 | O | TLV | 3-253 |
| 34 | Packet Flow Identifier | Packet Flow Identifier 10.5.6.11 | O | TLV | 3 |
| 39 | SM cause | SM cause 2 10.5.6.6a | O | TLV | 3 |
| B- | Connectivity type | Connectivity type 10.5.6.19 | O | TV | 1 |
| C- | WLAN offload indication | WLAN offload acceptability 10.5.6.20 | O | TV | 1 |
| 33 | NBIFOM container | NBIFOM container 10.5.6.21 | O | TLV | 3 – 257 |

## The Mobile Pwn2Own Bug (III)

By reversing the SM task, we find the handlers for the different messages.

One of these messages is the ACTIVATE PDP CONTEXT ACCEPT message.

One part of it that seems to be interesting is the Protocol Configuration Options, the function processing that IE seems complicated.

# The Mobile Pwn2Own Bug (IV)

```
signed int __fastcall sm_ProcessProtConfigOpts(unsigned __int8 *buf, unsigned int bufsize, unsigned __int8 *a3)
{
  ...
  unsigned __int8 v62[16]; // [sp+8h] [bp-58h]
  ...
  while ( idx < bufsize_ )
  {
    cursord = &buf__[idx];
    v10 = idx + 2;
    v11 = buf__[v10];
    idx = v10 + 1;
    proto = (unsigned __int16)(cursord[1] + (*cursord << 8));
    ...
    v61 = v13;
    ...
    if ( v11 )
    {
      if ( proto == 0x8021 )
      {
        subprot = buf__[idx];
        nidx = idx + 2;
        if ( subprot == 2 || subprot == 3 || subprot == 4 || subprot == 1 )
        {
          nptr = &buf__[nidx];
          idx = nidx + 2;
          v18 = (unsigned __int16)(nptr[1] + (*nptr << 8));
          if ( v18 >= 4 )
          {
            v19 = v18 - 4;
            while ( 2 )
            {
              v28 = v19;
              while ( 1 )
              {
                if ( !v28 )
                  goto LABEL_217;
                v20 = buf__[idx];
                v21 = idx + 1;
                if ( v20 != 3 )
                  break;
                v23 = buf__[v21];
                idx = v21 + 1;
                v24 = v23;
                ...
                if ( !len || len > 0x10 )
                {
                  if ( byte_41695FA4 )
                  {
                    v60 = 1108168344;
                    v61 = ((unsigned __int8)byte_41695FA4 << 18) + 0x40521;
                    v25 = len;
                  }
                  else
                  {
                    v25 = len;
                    v60 = 1108168372;
                    v61 = 0x40521;
                  }
                  DbgRelatedFcn_0(&v60, v25, 0xFECDBA98);
                }
                for ( i = 0; i < (signed int)(v24 - 2); i = (i + 1) & 0xFF )
                {
                  c = buf__[idx++];
                  v62[i] = c;
                }
                ...
              }
            }
          }
        }
      }
    }
    ...
}
```

## The Mobile Pwn2Own Bug (V)

Processes Protocol Configuration Options which are sent by the Network in a `ACTIVATE PDP CONTEXT ACCEPT`.

PDP stands for Packet Data Protocol

The purpose of the protocol configuration options information element is to:

- transfer external network protocol options associated with a PDP context activation, and
- transfer additional (protocol) data (e.g. configuration parameters, error codes or messages/events) associated with an external protocol or an application.

# The Mobile Pwn2Own Bug (VI)

One of the supported protocols is **IPCP** (Internet Protocol Control Protocol).

**IPCP header:**

| *Offsets* | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Code | | | | | | | | Identifier | | | | | | | | Length | | | | | | | | | | | | | | | |

**Code.**

8 bits.
Specifies the function to be performed.

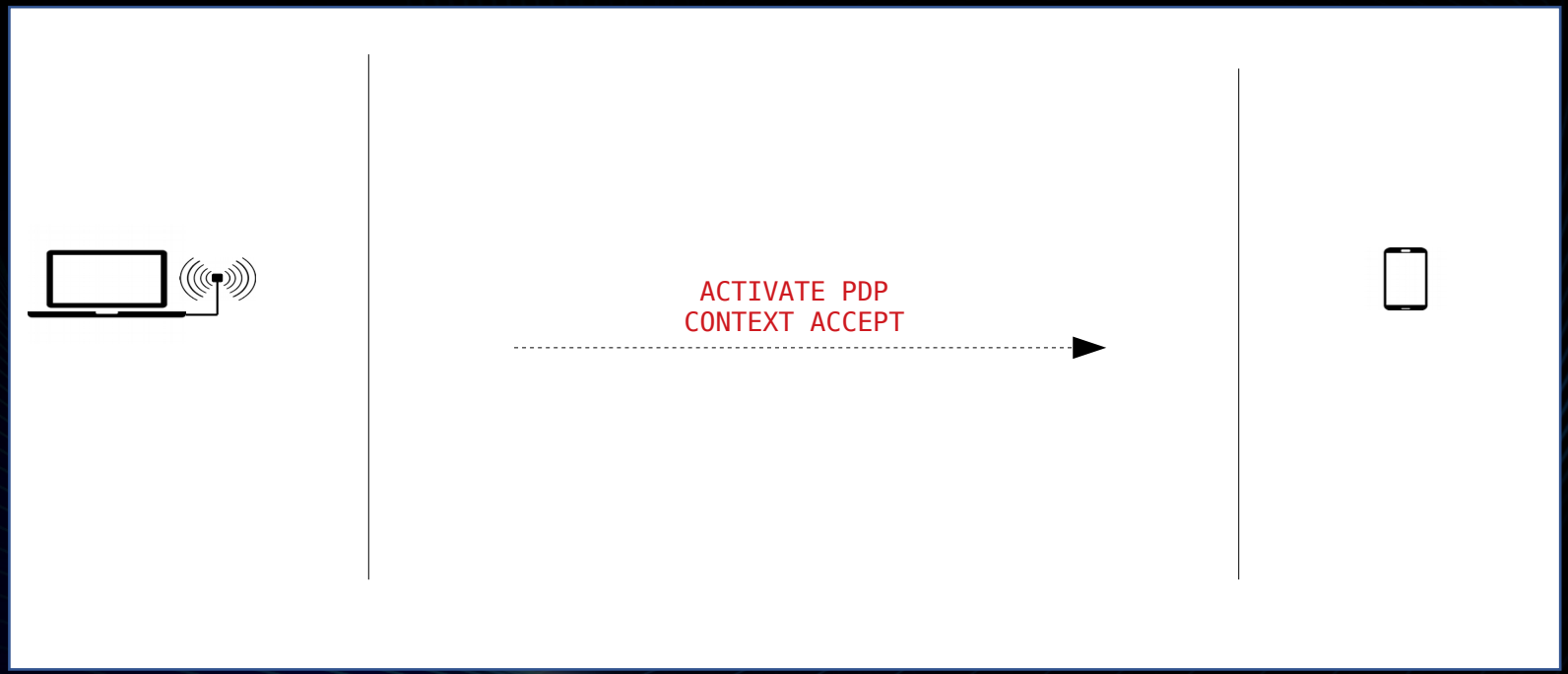
| Code | Description | References |
|---|---|---|
| 0 | Vendor Specific. | RFC 2153 |
| 1 | Configure-Request. | |
| 2 | Configure-Ack. | |
| 3 | Configure-Nak. | |
| 4 | Configure-Reject. | |
| 5 | Terminate-Request. | |
| 6 | Terminate-Ack. | |
| 7 | Code-Reject. | |

# The Mobile Pwn2Own Bug (VII)

```
int sm_ProcessProtConfigOpts(unsigned __int8 *buf, unsigned int bufsize, unsigned __int8 *a3)
{
    unsigned __int8 tbuf[16];
    int idx;
    int len;
    unsigned __int8 c;

    ...

    len = buf[idx++];
    ...
    if ( !len || len > 0x10 )
    {
        // Do nothing about it
    }
    for ( i = 0; i < len - 2; i = (i + 1) & 0xFF )
    {
      c = buf[idx++];
      tbuf[i] = c;
    }

    ...
}
```

## The Mobile Pwn2Own Bug (VIII)

The plan looks like this:



## The Mobile Pwn2Own Bug (IX)

Not so easy...

Problem is the phone will only process this message if it is in the correct state.

This happens when the phone sends a `ACTIVATE PDP CONTEXT REQUEST` message.

Which in turn happens if the phone is manually configured to include an APN in the connection settings.

However this is a problem for the P2O...

## The Mobile Pwn2Own Bug (X)

Read more of the technical standards...

We can force the MS get in the correct state (i.e perform PDP activation procedure) by sending a `REQUEST PDP CONTEXT ACTIVATION`.

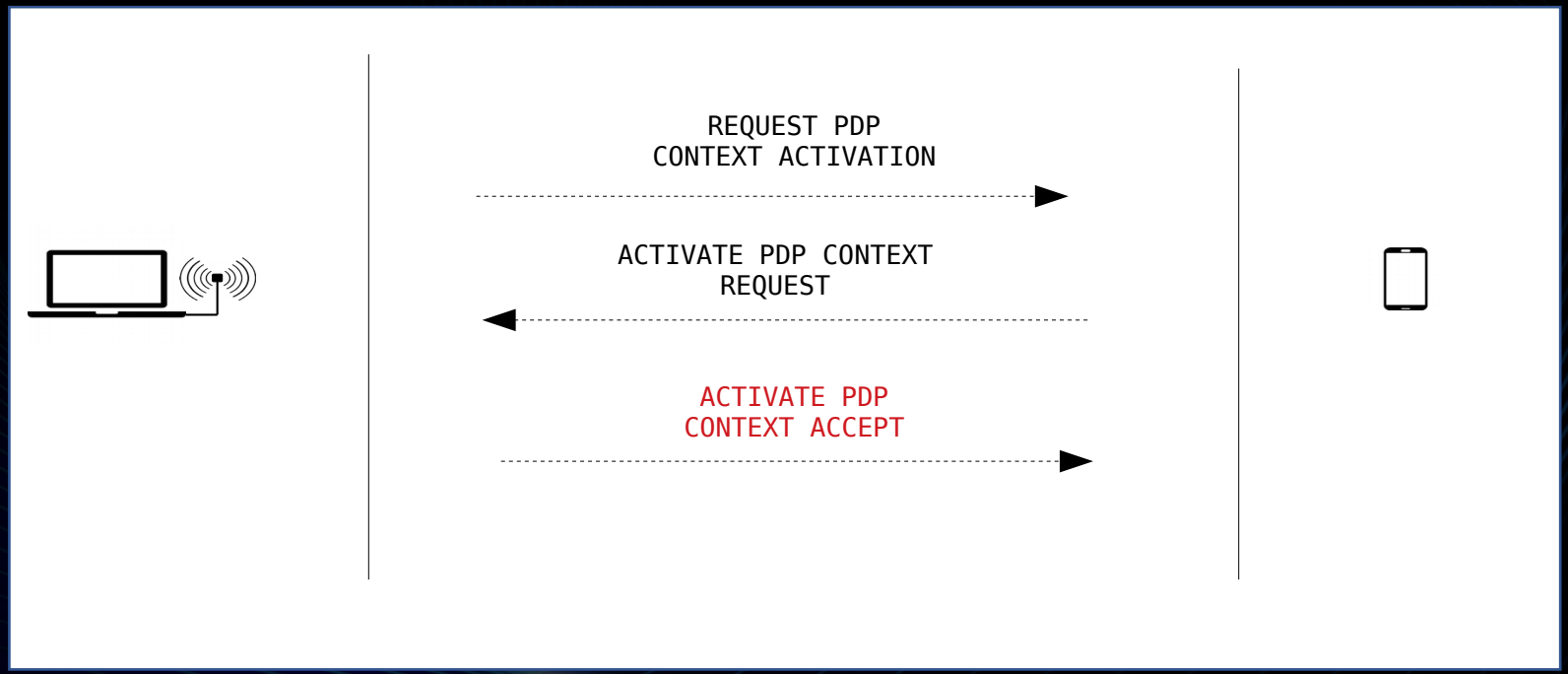
### 9.5.7 Request PDP context activation

This message is sent by the network to the MS to initiate activation of a PDP context.
See table 9.5.7/3GPP TS 24.008.

| | |
|---|---|
| Message type: | REQUEST PDP CONTEXT ACTIVATION |
| Significance: | global |
| Direction: | network to MS |

## The Mobile Pwn2Own Bug (XI)

The actual plan looks like this:



## The Mobile Pwn2Own Bug (XII)

ROP is needed for the first stage of your payload due to ARM cache-fu.

Copy shellcode to some arbitrary RWX address and invalidate/flush the i-cache/d-cache.

Jump to win.

Payload can do any number of things, for P2O I chose to write to the Android filesystem by leveraging the **RFS** (Remote? File System), a mechanism which allows the baseband to store data such as NV Items to the android filesystem.

Payload can even be a custom “debugger” that can be used to find other bugs and write more involved exploits (e.g heap memory corruption).

# Demo

# “Advanced” Debugging

## Custom Debugger (I)

As mentioned previously, we can use the shellcode as a debugger.

Modify .text to insert “breakpoints”.

```
1   void set_bp(void * addr, int t){
2       unsigned int * addr32 = (unsigned int *)addr;
3       unsigned short * addr16 = (unsigned short *)addr;
4       if(t){
5           addr32[0] = 0xf04f;
6           addr16[1] = 0x37b1;
7           addr16[2] = 0x603f;
8       }else{
9           addr32[0] = 0xe51fc000;
10          addr32[1] = 0xe58cc000;
11          addr32[2] = 0xb1b1b1b1;
12      }
13      flush_dache_range(addr, 0x100);
14      invalidate_icache_brpred();
15  }
```

## Custom Debugger (II)

Hook *malloc()* and *free()* and log to some file using *RFS*. Get trace from resulting crashdump.

```
1  void heap_tracer(){
2      // patch malloc() and free()
3      unsigned int ** addr = (unsigned int **)MALLOC_FUNC_PTR;
4      *addr = (unsigned int *)&hmalloc;
5      addr = (unsigned int **)FREE_FUNC_PTR;
6      *addr = (unsigned int *)&hfree;
7  }
8  void * hmalloc(int pool, int size, char * file, int line){
9      char tmpbuf[256];
10     unsigned int * res = 0;
11     res = malloc(pool, size, file, line);
12     snprintf(tmpbuf, sizeof(tmpbuf), "malloc(%d, %d, %s, %d) = 0x%p\n", pool, size, file, line, res);
13     log_msg(tmpbuf);
14     return res;
15 }
16 void hfree(void * paddr, char * file, int line){
17     char tmpbuf[256];
18     if(paddr){
19         unsigned int * addr = *((unsigned int **)paddr);
20         snprintf(tmpbuf, sizeof(tmpbuf), "free(*0x%p = 0x%p, %s, %d)\n", paddr, addr, file, line);
21         log_msg(tmpbuf);
22     }
23     free(paddr, file, line);
24 }
```

## Custom Debugger (III)

*free()* is called with: pointer to pointer, calling file and line number.

```
free(*0x41691278 = 0x437a05bc, ../../../HEDGE/GL3/GRR/Code/Src/rr_main.c, 1713)
free(*0x42efab98 = 0x43765820, ../../../HEDGE/NASL3/MM/Code/Src/mm_Utilities.c, 506)
free(*0x41691278 = 0x437a05a0, ../../../HEDGE/GL3/GRR/Code/Src/rr_main.c, 1713)
free(*0x41693eb0 = 0x437a023c, ../../../HEDGE/NASL3/MM/Code/Src/mm_Main.c, 609)
free(*0x4296d1b0 = 0x437ac03c, ../../../HEDGE/GL2/LLC/Code/Src/llc_Data.c, 5411)
free(*0x43010bb8 = 0x437a00bc, ../../../HEDGE/GL2/LLC/Code/Src/llc_Main.c, 173)
```

*malloc()* is called with: “pool” number, size, calling file and line number.

```
malloc(4, 13, e/Src/ns_OsInterface.c, 160) = 0x43799360
malloc(4, 13, e/Src/ns_MsgHandler.c, 321) = 0x43799460
malloc(4, 13, e/Src/ns_MsgHandler.c, 321) = 0x43799520
malloc(4, 4, ../../../HIU/MCD/HIC/SIPC/Code/Src/sipcCallCmdHandle.c, 3079) = 0x4376e5e0
malloc(4, 20, ../../../HIU/COMMON/HID/HostIF/Code/Src/hostifTransportMgr.c, 1978) = 0x43799360
malloc(4, 122, ../../../HIU/MCD/HIC/SIPC/Code/Src/sipcSSCmdhandle.c, 2231) = 0x43797420
```

## Custom Debugger (IV)

We can also use the RFS functionality mentioned above to build a debug server.

The client on the Application Processor and server on the Cellular Processor communicate through a file on the Android filesystem.

Can be used to implement basic functionality such as live memory inspection, breakpoint setting, code patching, etc.

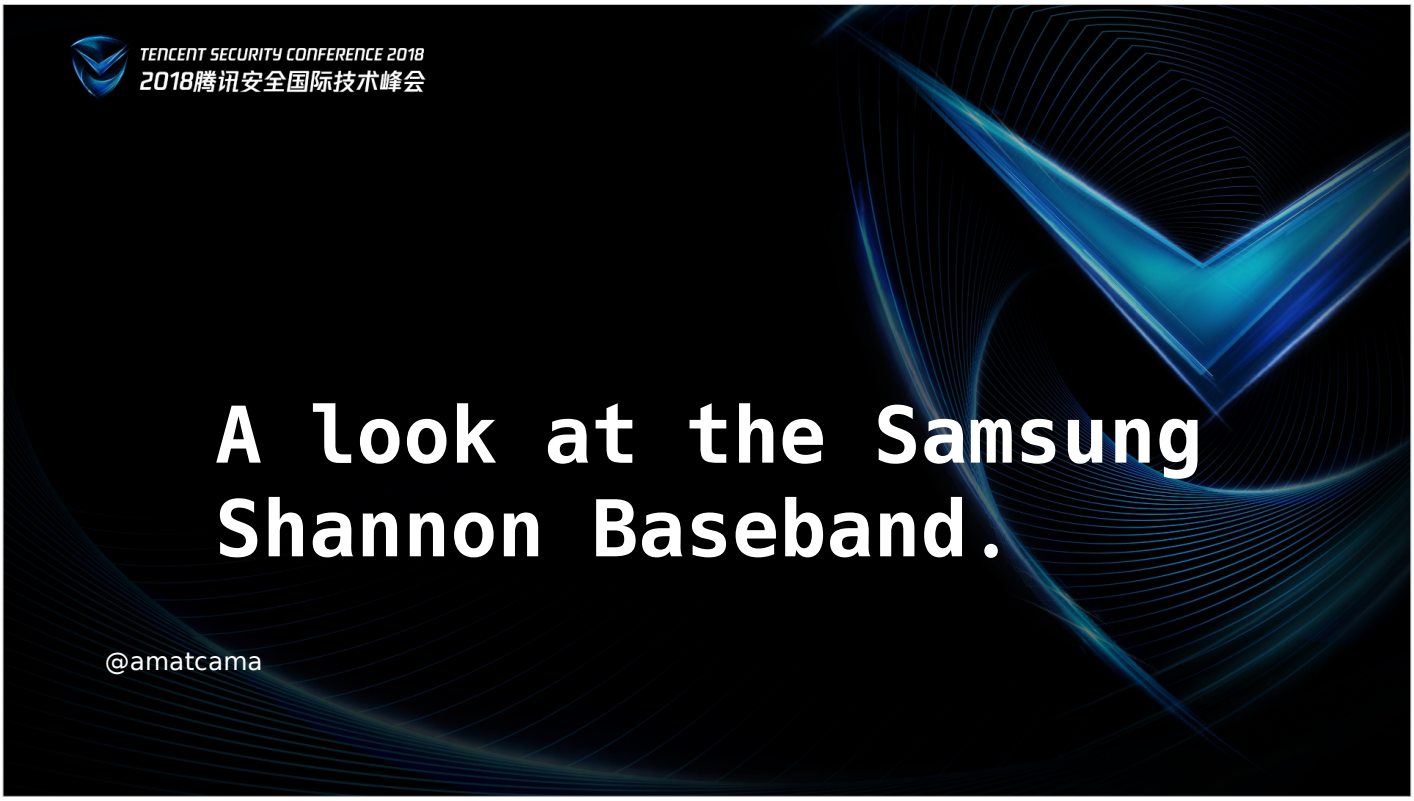
Can also be used to debug the interface between CP and AP by adding functionality to send arbitrary messages through the IPC channels.

# Conclusions

- Baseband exploitation isn’t as hard as it is percieved to be.
- You don’t need to know much about cellular networks in order to exploit them.
- The cellular protocol stacks are complicated enough that there should be many bugs still waiting to be discovered.
- Many targets out there, Mediatek, Huawei, Intel, Qualcomm...

?’s

acez@securin.tech


TENCENT SECURITY CONFERENCE 2018
2018腾讯安全国际技术峰会
A look at the Samsung Shannon Baseband.
@amatcama

**Amat Cama**

- Independant Security Researcher / **Securin Technology**
- CTF player @ Shellphish.
- Exploitation and Reverse Engineering.
- Currently interested in Hypervisors and Baseband reseach.

**Securin Technology**

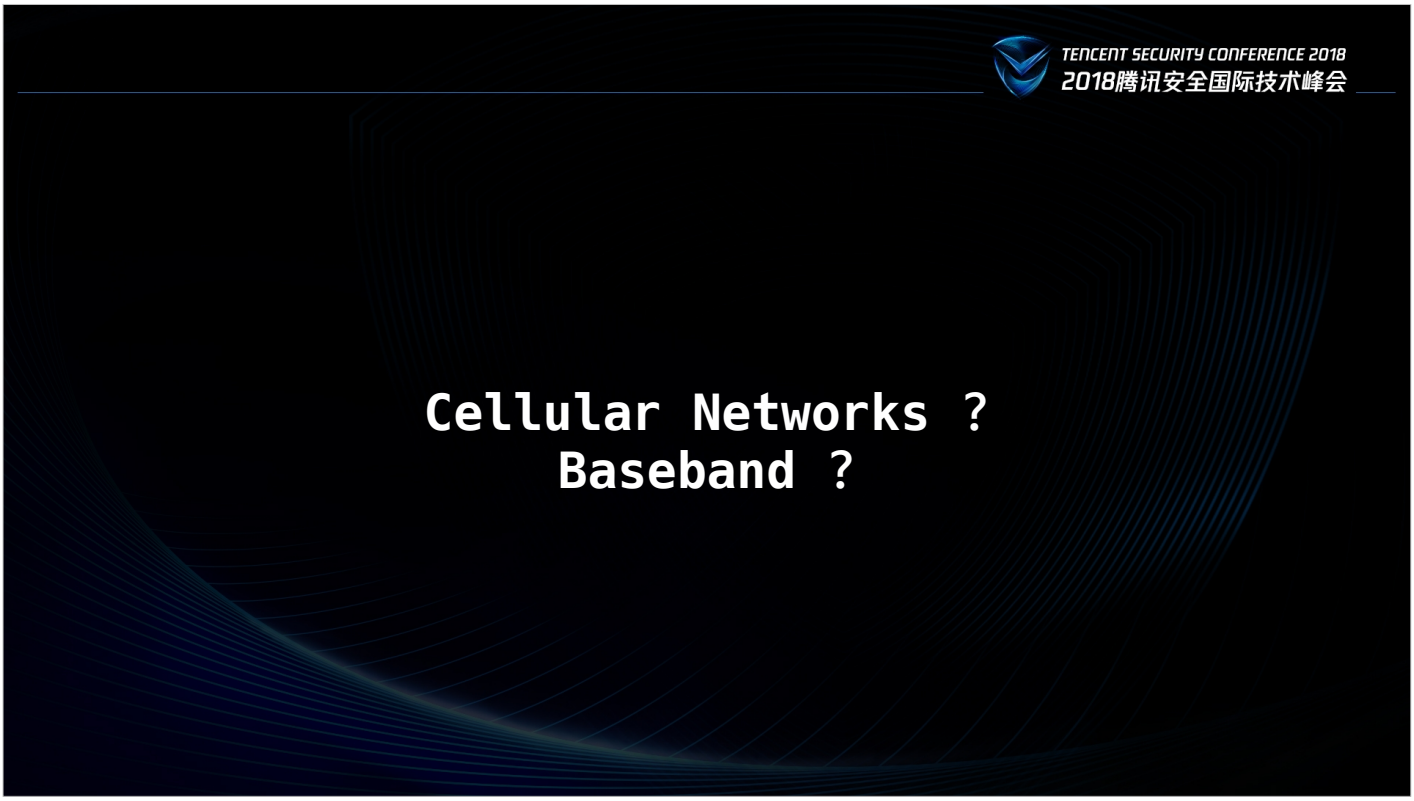
- Security Research.
- Trainings (Vulnerability Research, RE, Exploitation).
- Consultation (Code Audits, Vulnerability Assessments).

## Agenda

- Prior Work.
- Cellular Networks ? Baseband ?
- The Shannon Baseband.
- Hunting For Bugs.
- Demo.
- “Advanced” Debugging.
- Conclusions.

# Prior Work

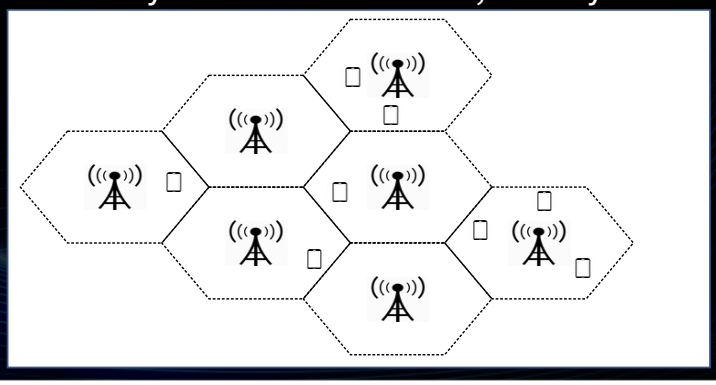
- “**Breaking Band - reverse engineering and exploiting the shannon baseband**” – *Nico Golde and Daniel Komaromy.*
- “***Exploitation of a Modern Smartphone Baseband***” – *Marco Grassi, Muqing Liu and Tianyi Xie.*




Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

## What is a Cellular Network ?

- Mobile communication network.
- “Cells” are land areas covered by a *base transciever station (**BTS**)*.
- To cover a large area, the cells are used in junction: *A Cellular Network*.
- Technically could be any kind of network, today mostly *Mobile Phone Network*.

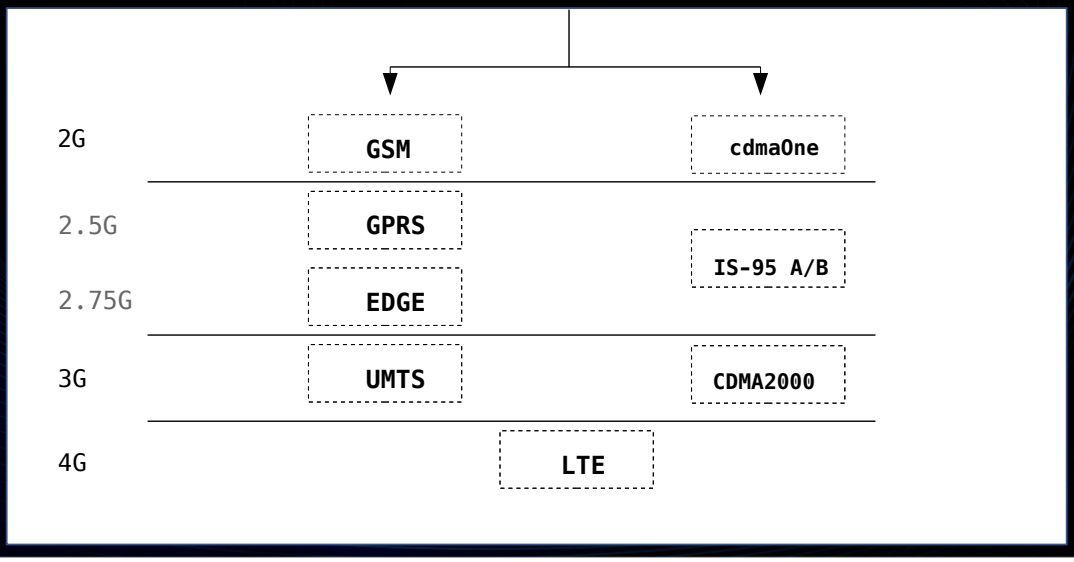


- Handoff between Cells as devices move around.
- Combination of all the different cells create a large network.


Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

## The technologies and standards (II)

- Mainly two branches: ***GSM*** branch and ***CDMA*** branch

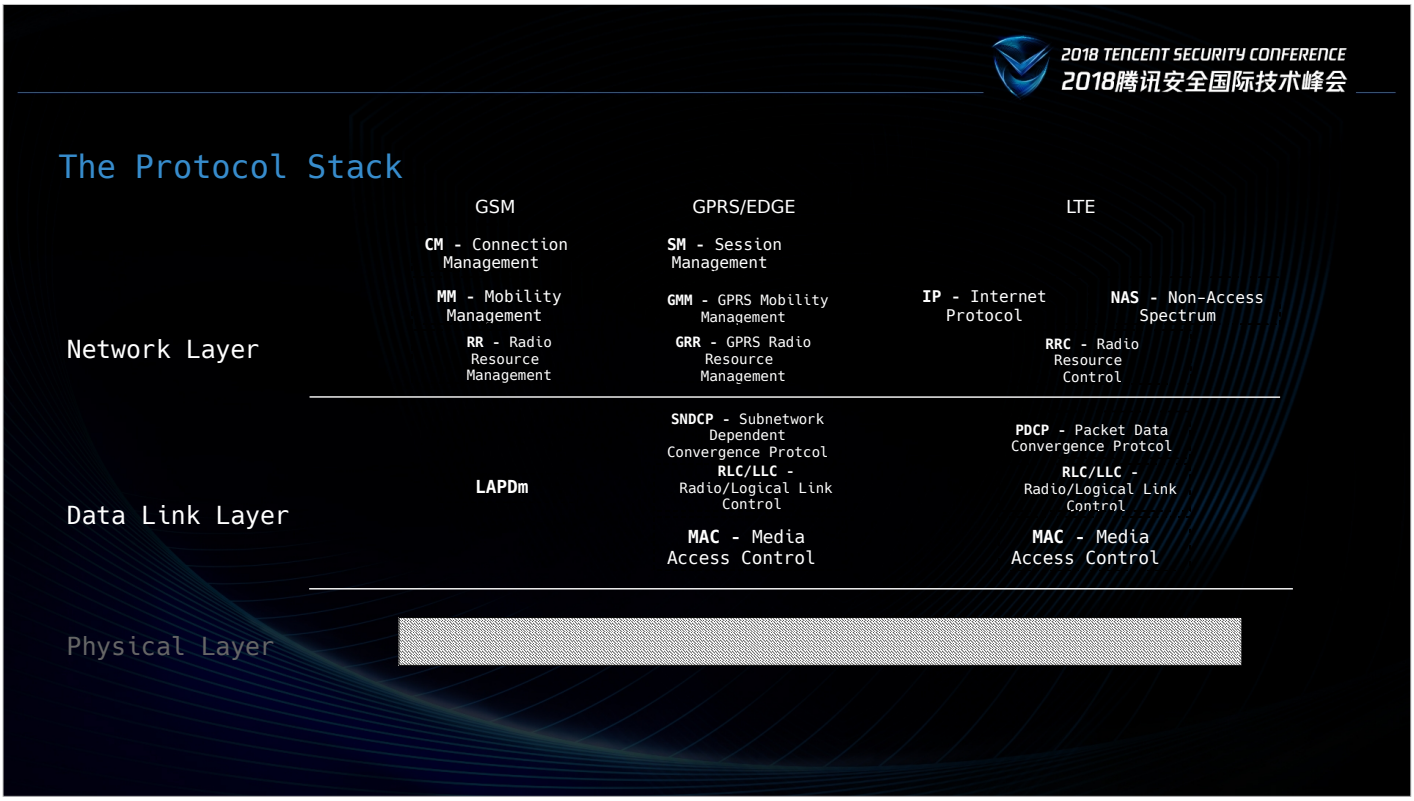



Different types of cellular phone networks

- 1G networks are analog cellular systems (1980s).
- 2G networks are the first digital cellular systems (1990s). Digital modulation techniques. GSM, cdmaOne
- 2.5G and 2.75G networks (GPRS/EDGE, IS-95B) are enhanced versions of 2G networks, with higher data rates.
- 3G networks (UMTS WCDMA FDD and CDMA2000) cellular networks have higher data rates and increased capacity.
- 4G is the current badass, soon to be superseeded by 5G.
- **This talk will mainly focus on the GSM branch but it is good to note that CDMA is also a good branch to look, KeenLab folks demoed some vulnerabilities in CDMA code.**

## The technologies and standards (III)

- *3GPP* is a collaboration agreement with a number of telecommunication standard bodies.
- Provides maintenance and development of the GSM *Technical Specifications (TS)*
  - GSM
  - GPRS / EDGE
  - UMTS
  - LTE
- Is Comprised of bodies such as the *European Telecommunications Standards Institute (ETSI)*.
- The technical standards provide detailed information on the structure of messages exchanged.




Scary slide but just a reference :)

- This diagram is inaccurate, but I think it is a good way to think about the stacks
- Abstract a lot of other intricacies that we don't care about for this talk.
- As you dig deeper into the technologies you will see how wrong it is but is a good starting point.
- 3G/UMTS is somewhere between GPRS/EDGE and LTE ~~ abouts
- Assume that the different technologies are non-exclusive, especially GSM and GPRS (because GPRS is an extension of GSM).
- CM: Call Control (establishment etc.), SMS handling amongs others
- MM: Mobility, authentication and security
- RR: link management between phone and network, allocation of dedicated channels.
- SM: session establishment and management for packet data transfers.
- LTE NAS: has EMM and ESM which are equivalents of the GMM and SM components in GSM/GPRS

## The Baseband (I)

- Component of the phone in charge of handling communication with the mobile network.
- Deals with low level radio signal processing.
- Supports a number of standards (GSM, 3G, 4G, 5G, cdmaOne, CDMA2000, ...).
- Basically the main “interface” to the mobile network.

## The Baseband (II)

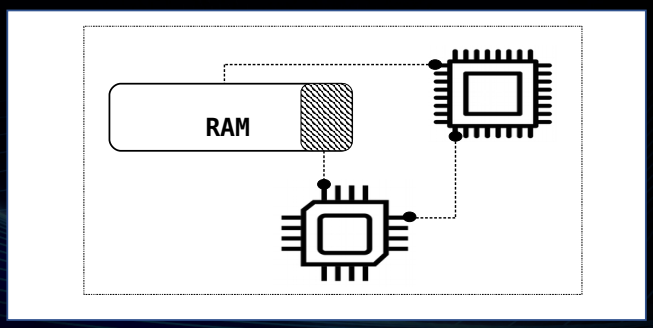
- A number of different implementations.
- Qualcomm owns most of the market.
- Qualcomm: Galaxy, iPhone, OnePlus, Pixel, Xperia, HTC, LG, ASUS, Motorola, ...
- Huawei: Mate 10, P20, Honor 9, ...
- **Samsung**: Galaxy S6, S7, S8, S9, ...
- Intel: iPhone Xs/r,iPhone X, iPhone 8, ...
- Mediatek: Xiaomi, Tecno, ...

Qualcomm is losing its market dominance, in fact not sure if it is accurate to say that it still owns most of the market

Mediatek is very popular in Africa, a lot of low-cost smart phones use the mediatek Helio SoC which comes with its own baseband.

## The Baseband (III)

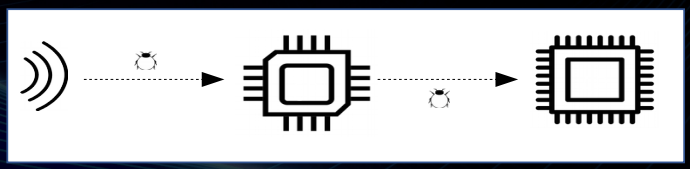
- he most common architecture today: baseband firmware runs on a dedicated chip; the *cellular processor (**CP**)*.
- This chip is tasked with all of the radio processing.
- The code is generally written in low level languages such as C/C++.
- A communication interface between **CP** and **AP** (Application Processor) such as shared memory, serial or interrupts.




AP: Application Processor runs all other android things, your browser and all those other applications that aren’t really part of a phone.

## The Baseband (IV)

- Getting code execution on the *CP* doesn't necessarily result in owning the whole device.
- A number of attacks can be performed:
  - → Redirect/Intercept phone calls.
  - → Redirect/Intercept SMS.
  - → Modify Internet traffic.
  - → ...
- A step further; attack the *AP* through the IPC mechanisms and gain full control of the device.



- However it is not impossible to reach endpoints that are directly running on the Application processor for example some SMS data.


Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

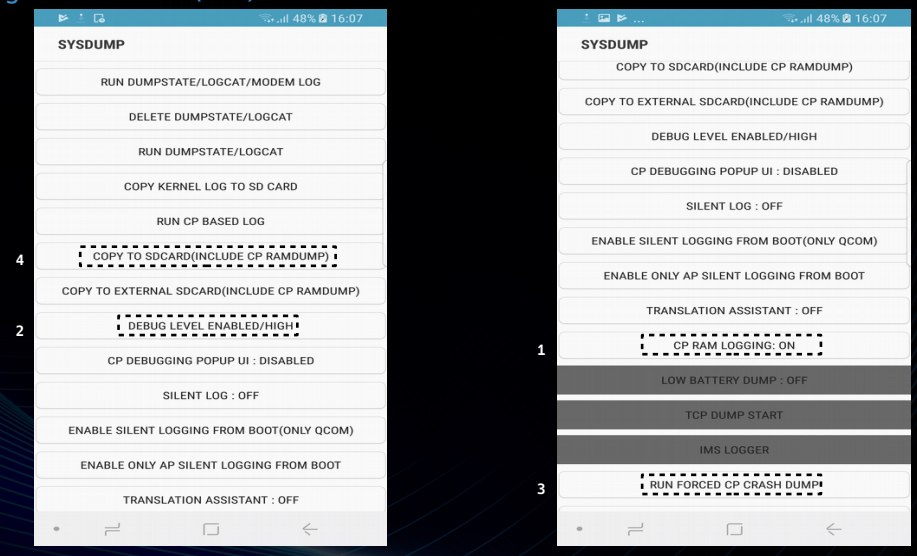
## About Shannon

- Samsung’s Baseband implementation.
- Typically ships with phones featuring the Exynos SoC.
- e.g: most non-US Galaxy phones.
- A RTOS running on an ARM Cortex R7.

## Obtaining the code (I)

- The modem firmware can be obtained from the phone's firmware images.
- However it is encrypted and doesn't seem to be an easy way to decrypt it.
- Luckily it is possible to make the phone generate modem RAM dumps.
- Dialing the code *#9900# brings up the *SYSDUMP* menu.

## Obtaining the code (II)



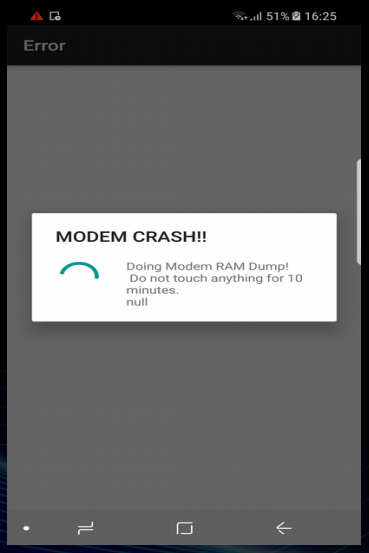

- Debug Level: Set debug level to high for verbosity.
- CP RAM Logging: Set to on, to make phone generate modem RAM dumps everytime the modem crashes.
- Run Force CP Crash DUMP: Force a modem crash.
- Copy To Sdcard: Copy the modem RAM dump to a readable location because it is stored in a location on the filesystem that the adb user can’t read.

## Obtaining the code (III)

- Tap on the `DEBUG LEVEL ENABLED/` option and set it to `High`. The phone will reboot.
- Reopen the SYSDUMP menu, scroll down and tap on the `CP RAM LOGGING` option and set it to `On`. The phone will reboot.
- Reopen the SYSDUMP menu and scroll all the way down, tap the `RUN FORCED CP CRASH DUMP` option. The phone will reboot and go into the ram upload mode. Hold the power and volume down button for 10 seconds to turn the phone off and then power it back on.
- Reopen the SYSMDUMP menu and tap the `COPY TO SDCARD(INCLUDE CP RAMDUMP)` option.
- Now in the folder `/sdcard/log` of the device, we have the log files including the ram dump. Largest file in the folder and has a name of the following format `cpcrash_dump_YYYYMMDD_HHSS.log`

This slide just describes the process for enabling modem ram dumps, it is not really meant to be described in the talk.

## Obtaining the code (IV)




On the left is the screen that is shown after the modem crashes while the phone is dumping the RAM.

On the left is the screen you get after the phone powers off due to a Modem crash (when RAM logging is enabled).

## Loading Code in IDA

- The CP Boot Daemon (/sbin/cbd) handles powering on the modem and processing RAM dumps amongst other things.
- Boot code can be found at the start of the encrypted modem image in the firmware packages.
- By reversing the cbd and boot, we can translate the file offsets of the RAM dump to virtual addresses:

```
0x40000000
0x8000000
0x4000000
0x20000
0x4800000
...
```

This part is described pretty well in the *Breaking Band* talk mentioned earlier.

They provide a script to load the RAM dump in IDA.

## Identifying Tasks

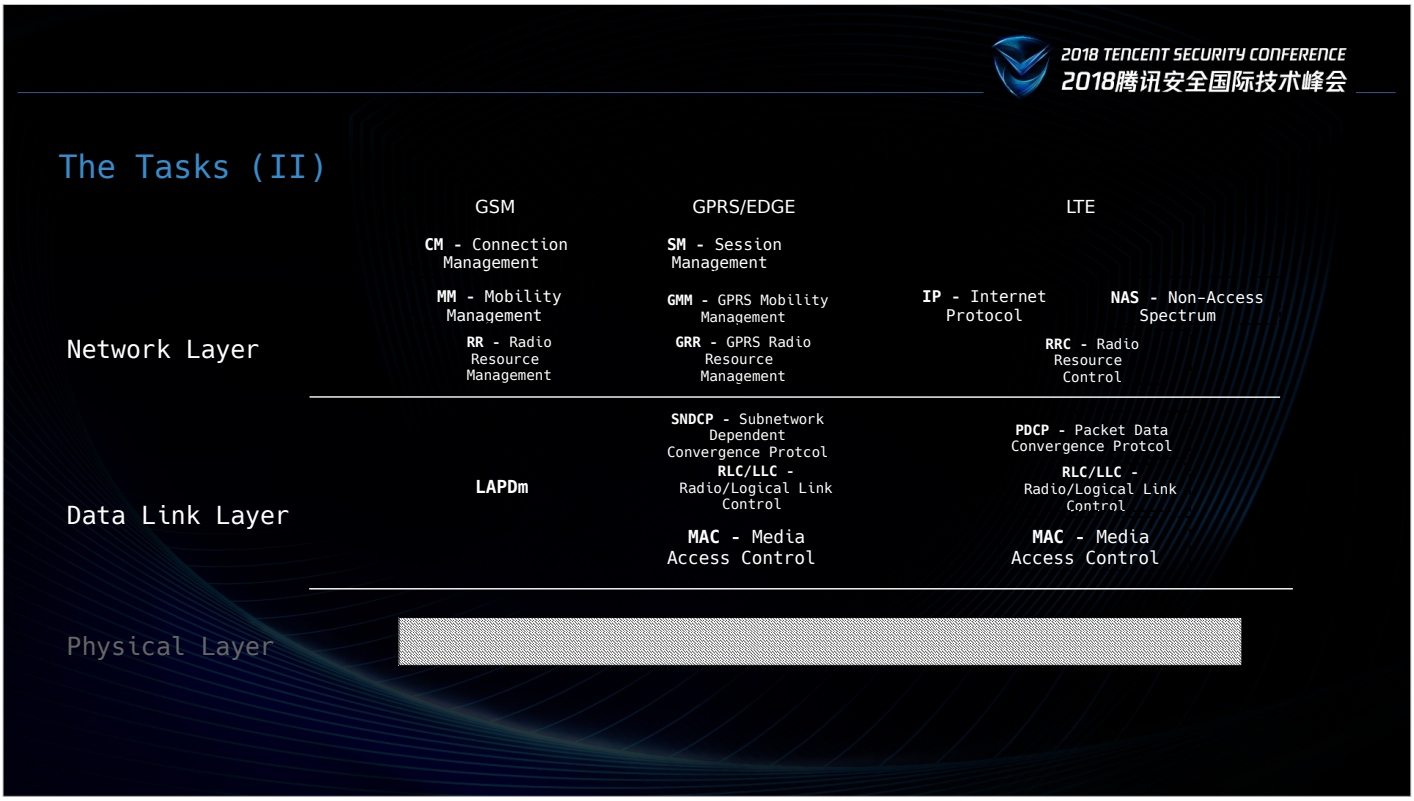
- We need to identify the different tasks run by the RTOS.
- Start reversing from RESET Exception Vector Handler…
- Look at the start of the different memory regions and you recognize the Exception Vector Table in one of them.
- A linked list contains all the different tasks' entry points, corresponding stack frames and task names (very useful).
- Traverse the list and identify all the tasks.

Again, the *Breaking Band* describes this process pretty well.

## The Tasks (I)

- We end up with a list of tasks with different names, some of them self-explanatory, some of them misleading, some of them hard to understand.
- **MM** (**M**obility **M**anagement ?)
- **SAEL3** (**S**ystem **A**rchitecture **E**volution / LTE **L**ayer **3** ?)
- SNDCP
- **CC** (**C**all **C**ontrol ?)
- **SM** (**S**ession **M**anagement ?)
- LLC
- ...

Different tasks are used for the different protocols and layers of the cellular technologies. This makes it a bit easier to reverse the code because we can just jump to these tasks and have an idea what kind of data they **should** be processing.

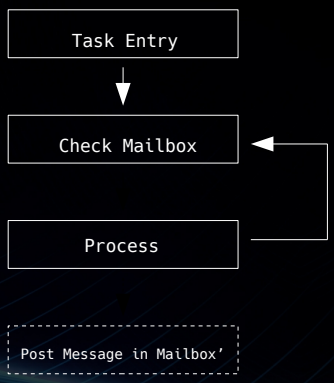



Scary slide but just a reference :)

- This diagram is inaccurate, but I think it is a good way to think about the stacks
- Abstract a lot of other intricacies that we don't care about for this talk.
- As you dig deeper into the technologies you will see how wrong it is but is a good starting point.
- 3G/UMTS is somewhere between GPRS/EDGE and LTE ~~ abouts
- Assume that the different technologies are non-exclusive, especially GSM and GPRS (because GPRS is an extension of GSM).
- CM: Call Control (establishment etc.), SMS handling amongs others
- MM: Mobility, authentication and security
- RR: link management between phone and network, allocation of dedicated channels.
- SM: session establishment and management for packet data transfers.
- LTE NAS: has EMM and ESM which are equivalents of the GMM and SM components in GSM/GPRS

## The Tasks (III)

- Different tasks are used for different components and layers of the protocol stacks.
- Tasks communicate with each other using a *mailbox* system.
- Tasks are pretty much *while* loops waiting to process messages (from other tasks).




For example, the RR task will do its thing and pass the message up to the MM task etc.

The fact that the tasks are basically just loops makes it easy to achieve process continuation after exploiting a vulnerability in them. Just jump back to the start of the loop after fixing up the stack and registers.

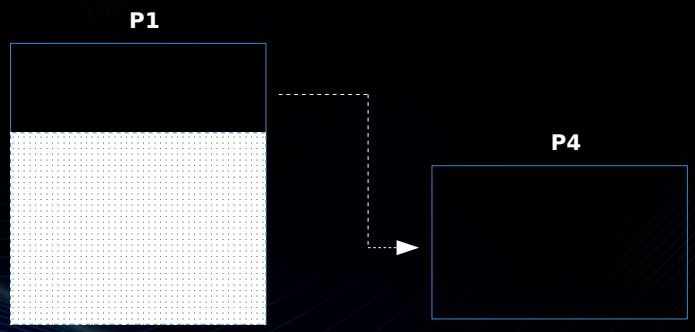
## The Tasks (IV)

- Pick a task and start reversing.
- The code is pretty generous in that it contains a **lot** of strings.

For example, the RR task will do its thing and pass the message up to the MM task etc.

## The Heap (I)

- The heap is segmented into different “pools” numbered **1** through **6**.
- Most of the allocations happen from pool number **4** (*the main pool*).
- The *main pool* is a client of pool **1**, 0x700000 bytes is allocated for it.




For example, the RR task will do its thing and pass the message up to the MM task etc.

## The Heap (II)

- The allocation and de-allocation functions are:
  - **pal_MemAlloc**(int pool_no, size_t alloc_size, char * filename, int line_no)
  - **pal_MemFree**(void **ptr, char * filename, int line_no)

Both the functions are called with the name of the file the call happens in, as well as the line number.

Please note that pal_MemFree() is called with a pointer to the pointer that wants to be freed.

## The Heap (III)

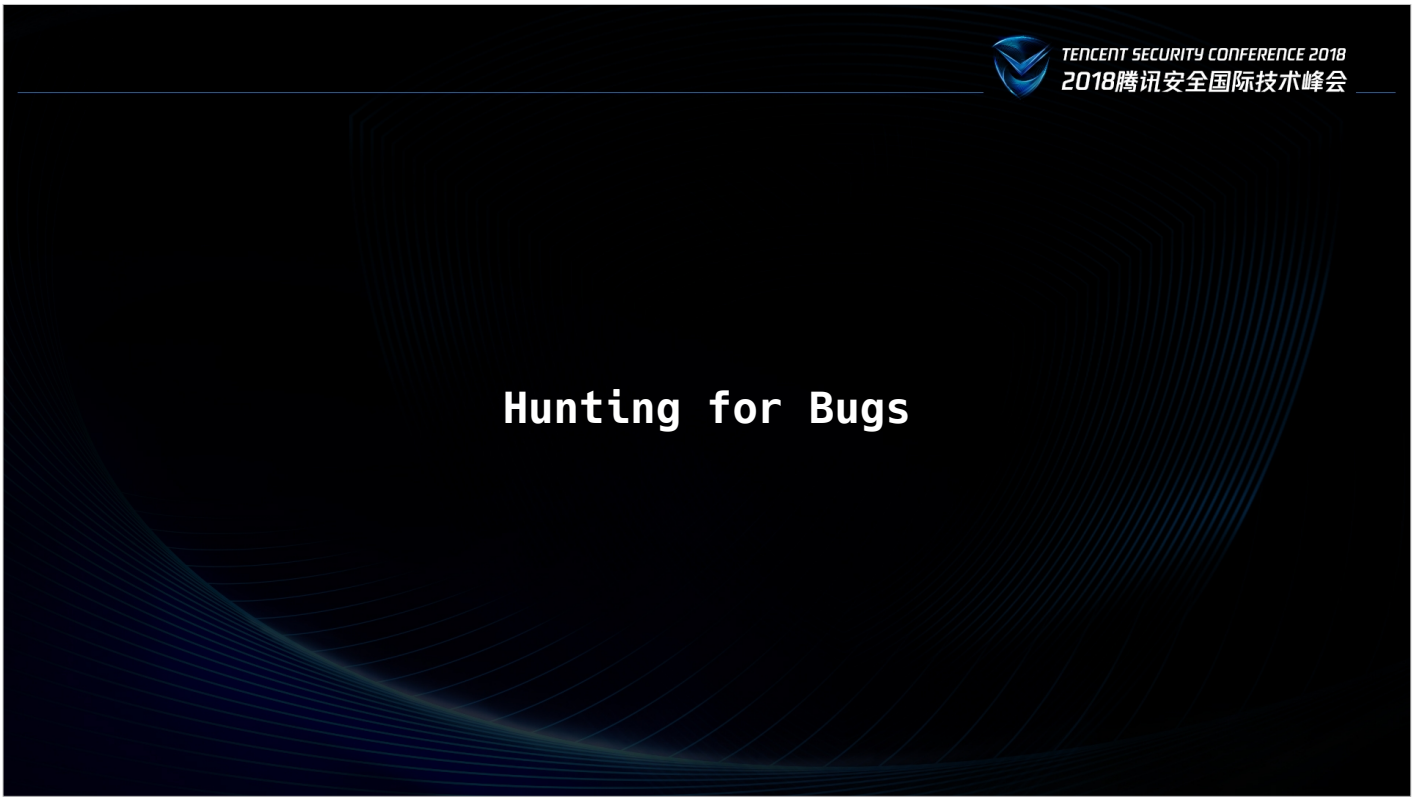
- An allocated heap chunk from the main pool has a header of size **0x20** bytes:


- The data is encircled by 0xAAAAAAAA which is a “memory guard” that is used to check for overflows during the free’ing of the chunk.
- The pool number is used in during free’ing in order to retrieve the pool this chunk is associated with.

## The Heap (IV)

- The pool is segmented into “pages” for different chunk sizes
- The 0x700000 bytes allocated for the main pool contain the pool header at the start:
  - A *control structure* describing free and used chunks (bitmap)
  - The *size* of the pool
  - The *start* and *end* address of the pool

- The start and end address of the pool are used during free’ing to make sure that the address passed are in the bounds of the heap chunk.
- Objects of the same (aligned) sizes end up in the same pages.
- The address of the object is used to compute an offset into the control structure during free’ing.
- As can be seen, the fact that the chunk information is kept out-of-band (in a seperate control structure) makes attacks on metadata not so easy.




Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

## Setting up an environment (I)

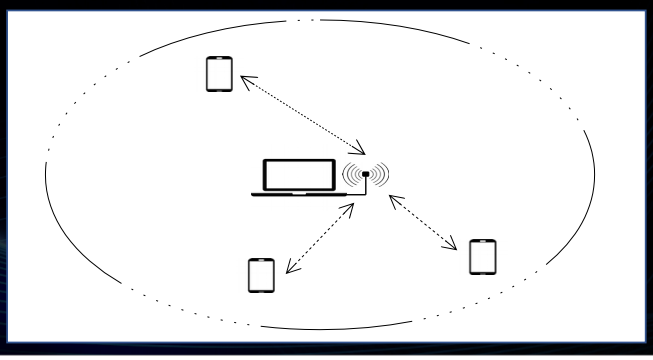
The goal is to be able to send arbitrary data the the baseband.

Need to operate our own cellular network.

Can be achieved with a *Software Defined Radio (**SDR**)*.

The Mobile Network Stack / Standard is implemented in software that runs on our computers.

The SDR (device) is a general purpose transciever that supports different frequencies.



## Setting up an environment (II)

A number of different options for the SDRs.

- *BladeRF x40*: $420.00
- ***BladeRF x115***: $650.00
- ***USRP B200***: $675.00
- ***LimeSDR***: $300.00
- *UmTRX*: $950.00 - $1300.00

The USRP is very stable.

## Setting up an environment (III)

A number of different options for software implementation of the standards.

***YateBTS:***

- Clean code, easy to modify.
- Supports *bladeRF*.
- GSM and GPRS.
- Easy to compile and run.

***OpenBTS*** *(OpenBTS-UMTS):*

- Clean code, easy to modify.
- Good support for *USRP* and *UmTRX*.
- GSM, GPRS, 3G.
- Easy to compile and run.

YateBTS is probably my favorite GSM stack.

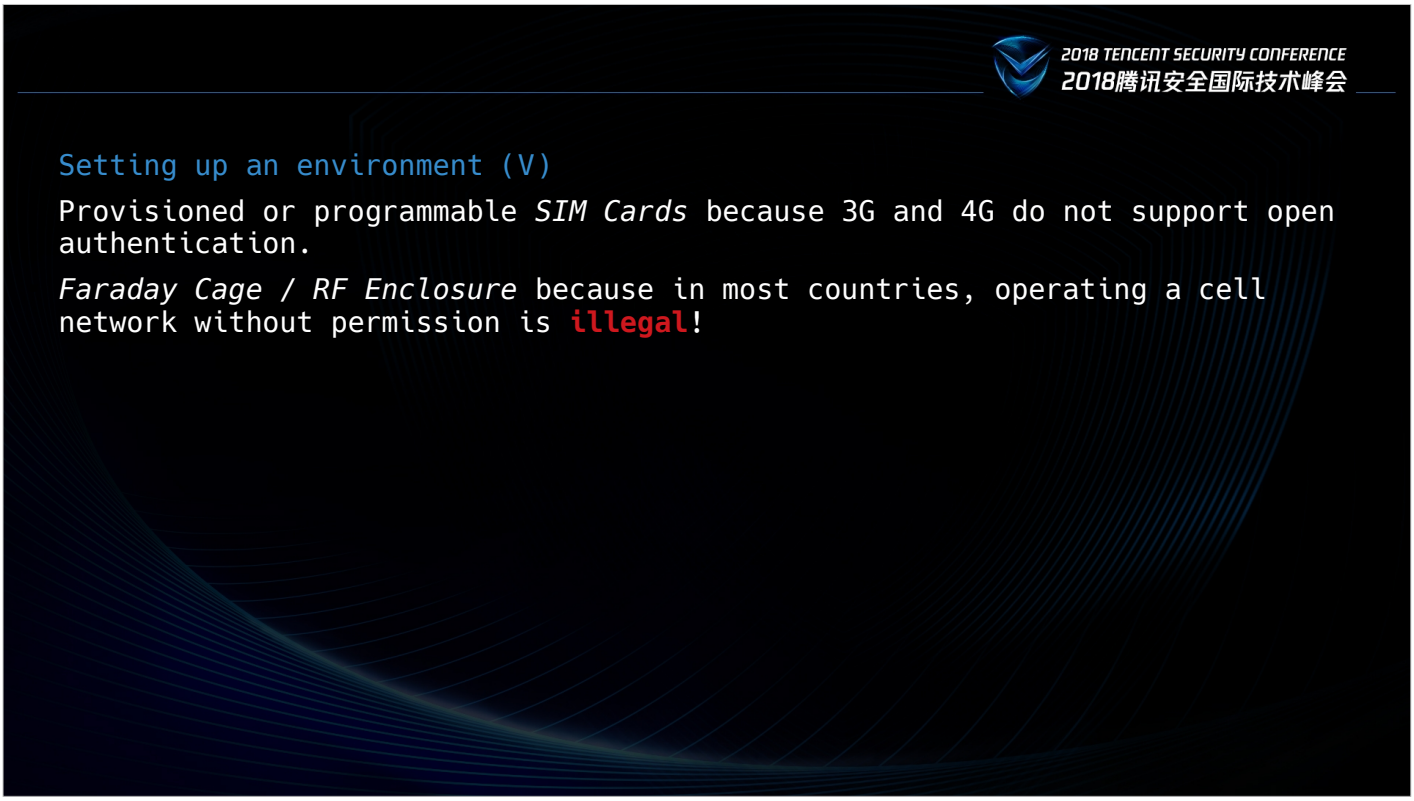
## Setting up an environment (IV)

*OpenBSC* (OsmoNITB, OsmBTS, …):
- Good support for *USRP*, *LimeSDR* and *UmTRX*.
- Compiling wasn't easy.
- Clean code, easy to modify.
- GSM + GPRS.

*OpenAirInterface:*
- Hard to compile and run.
- Good support for *USRP*.
- 4G/LTE.

***srsLTE:***
- Very easy to compile and run.
- 4G/LTE.
- Good support for *USRP* and *LimeSDR*.
- Clean code, easy to modify.

srsLTE is my favorite LTE stack.




- SIMs: Also it is easier to make a phone connect to your own network if you can program the SIMs with custom MCC and MNC.

## Debugging The Phone

Everytime the modem crashes we get a RAM dump.

Luckily the dump contains the state of the registers at the time of the crash, therefore we have pretty decent post-mortem debugging capabilities.

Write a script to process the dumps and do useful stuff (registers, peeking at memory).

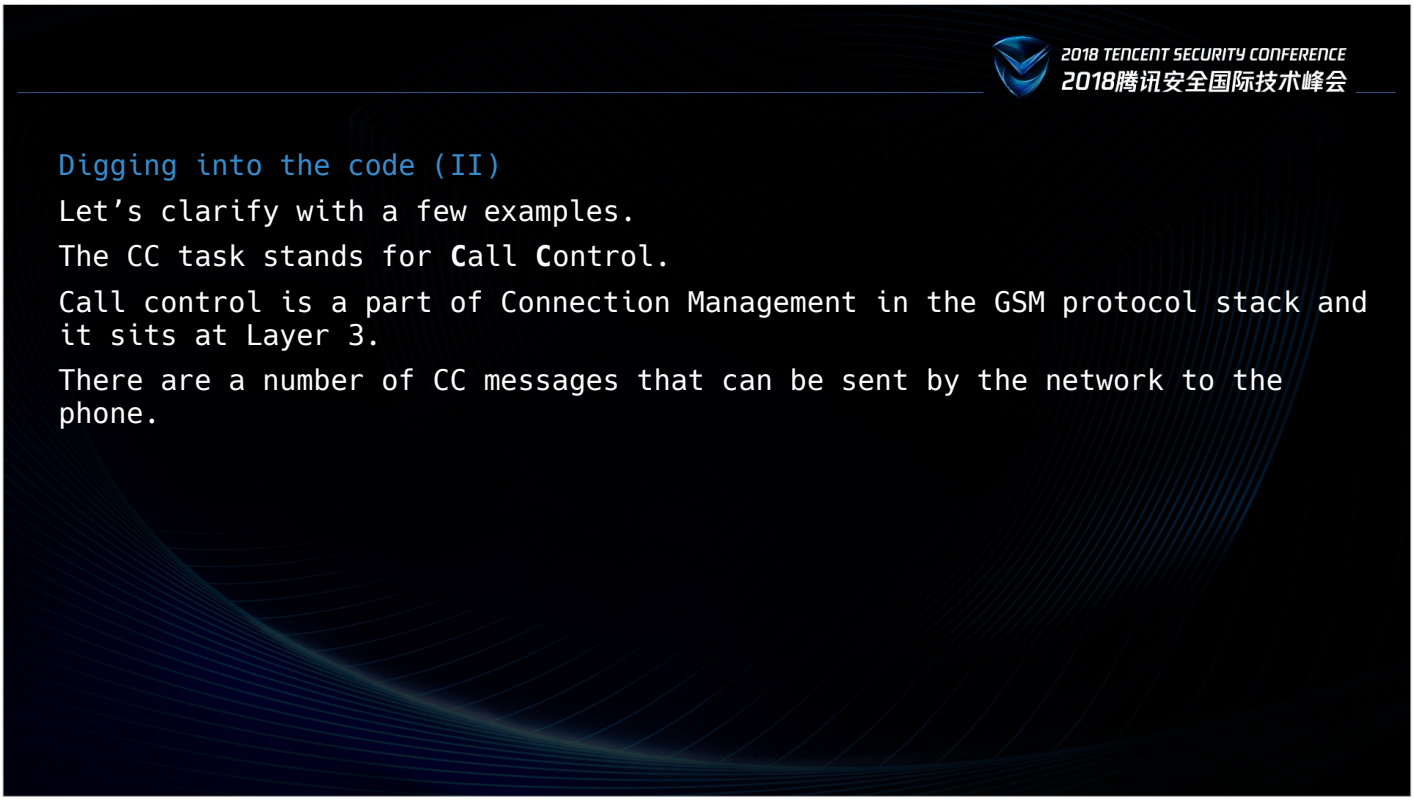
## Digging into the code (I)

Back to picking a task to have a closer look at.

Interesting area to look at is the Layer 3 / NAS:

- Layer 3 Messages are comprised of *Information Elements (IEs)*.
- IEs are *V*, *LV*, *TLV*.
- We can imagine code patterns that trust the length field without checks.

Samsung has been fixing a number of bugs this year.




- SIMs: Also it is easier to make a phone connect to your own network if you can program the SIMs with custom MCC and MNC.

## Digging into the code (III)

There are a number of CC messages that can be sent by the network:

- Alerting
- Call proceeding
- Connect
- Connect acknowledge
- Disconnect
- Facility
- Notify
- **Progress**
- Setup
- **Status**
- ...

- SIMs: Also it is easier to make a phone connect to your own network if you can program the SIMs with custom MCC and MNC.

## Progress Message Bug

**Table 9.67/3GPP TS 24.008: PROGRESS message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Call control protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2 |
| | Progress message type | Message type 10.4 | M | V | 1 |
| | Progress indicator | Progress indicator 10.5.4.21 | M | LV | 3 |
| 7E | User-user | User-user 10.5.4.25 | O | TLV | 3-131 |

Here we see that the prograss message has LV and TLV IEs.

The function parsing the Progress Indicator LV was vulnerable to a straight up stack buffer overflow.

Reported by Comsecuris.

## Status Message Bug

**Table 9.74/3GPP TS 24.008: STATUS message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Call control protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2 |
| | Status message type | Message type 10.4 | M | V | 1 |
| | Cause | Cause 10.5.4.11 | M | LV | 3-31 |
| | Call state | Call state 10.5.4.6 | M | V | 1 |
| 24 | Auxiliary states | Auxiliary states 10.5.4.4 | O | TLV | 3 |

Here we see that the Status message has LV and TLV Ies.

The function parsing the Cause LV was vulnerable to a straight up stack buffer overflow.

Reported by me.

## Digging into the code (V)

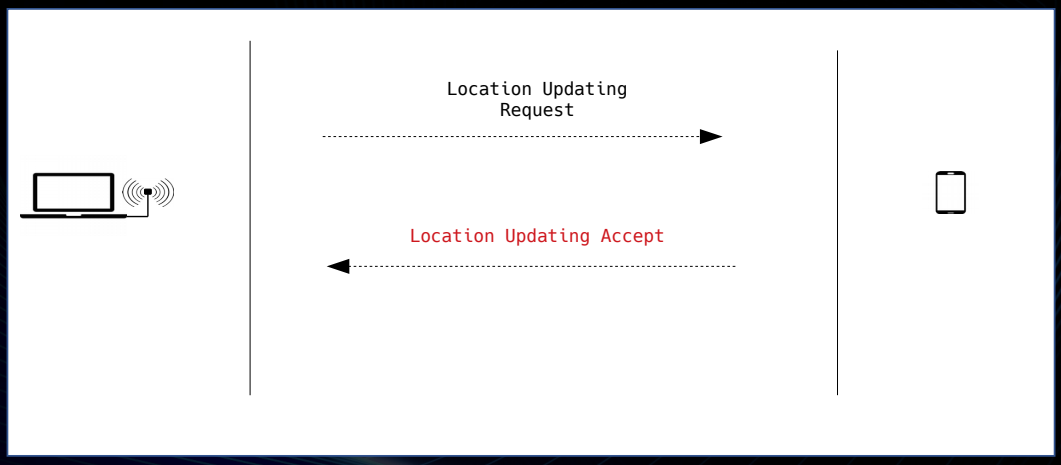
These vulnerabilities are pretty trivial. Essentially doing this:

```
1  void vuln(unsigned __int8 *tlv)
2  {
3      unsigned __int8 tbuf[N];
4      unsigned __int8 len;
5
6      ....
7
8      len = tlv[1];
9      ....
10
11     memcpy(tbuf, &tlv[2], len)
12     ...
13 }
```


- SIMs: Also it is easier to make a phone connect to your own network if you can program the SIMs with custom MCC and MNC.

## Location Updating Accept Bug

When the phone first connects to the network:




After connecting to the network, the phone sends a location updating request message, the network replies with a location updating accept message containing some information such as the mobile identity and emergency number list.

## Location Updating Accept Bug

**Table 9.2.15/3GPP TS 24.008: LOCATION UPDATING ACCEPT message content**

| IEI | Information element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Mobility management protocol discriminator | Protocol discriminator 10.2 | M | V | ½ |
| | Skip Indicator | Skip Indicator 10.3.1 | M | V | ½ |
| | Location Updating Accept message type | Message type 10.4 | M | V | 1 |
| | Location area identification | Location area identification 10.5.1.3 | M | V | 5 |
| 17 | Mobile identity | Mobile identity 10.5.1.4 | O | TLV | 3-10 |
| A1 | Follow on proceed | Follow on proceed 10.5.3.7 | O | T | 1 |
| A2 | CTS permission | CTS permission 10.5.3.10 | O | T | 1 |
| 4A | Equivalent PLMNs | PLMN list 10.5.1.13 | O | TLV | 5-47 |
| 34 | Emergency Number List | Emergency Number List 10.5.3.13 | O | TLV | 5-50 |
| 35 | Per MS T3212 | GPRS Timer 3 10.5.7.4a | O | TLV | 3 |

## Location Updating Accept Bug

The location updating accept message contains some information such as the *Mobile Identity* and *Emergency Number List*.

> "The purpose of the *Emergency Number List* information element is to encode emergency number(s) for use within the country where the IE is received".
>
> It has a maximum length of **50** bytes.

## Location Updating Accept Bug

```
1  int ParseEmergencyNumberList(unsigned __int8 *buf, unsigned int len, EmergencyNumber_t *output)
2  {
3    int idx;
4    unsigned int i;
5    int8_t emnumlen;
6
7    idx = 0;
8    for ( i = 0; i < len; i = i + emnumlen + 1) )
9    {
10     emnumlen = buf[i];
11     if ( !buf[i] )
12       break;
13     ParseEmergencyNum(buf[i], &buf[i + 1], &output[idx]);
14     idx = idx + 1;
15   }
16   return idx;
17 }
```

The bug here is a little more subtle, but it is still an issue of trusting the length field of the TLV.

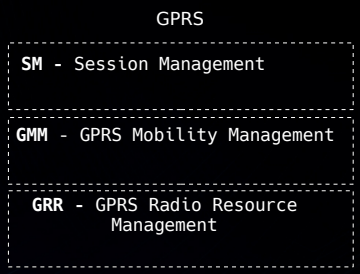
The calling function passes a pointer to an array of EmergencyNumber_t structures but there is no check on the total number of EmergencyNumber_t structures that can be written to it.

One thing to note is that this function is also reachable in the GPRS path during the attach procedure. The “Attach Accept” message also contains the Emergency Number List Information Element.

## The Mobile Pwn2Own Bug (I)

Decided to look at GPRS since it seems complicated?

Start by reading the standards and looking at the GPRS Session Management Messages.




SM because we assume that there shouldn’t be much difference between GMM and MM.

## The Mobile Pwn2Own Bug (II)

The ACTIVATE PDP CONTEXT ACCEPT message looks good.

**Table 9.5.2/3GPP TS 24.008: ACTIVATE PDP CONTEXT ACCEPT message content**

| IEI | Information Element | Type/Reference | Presence | Format | Length |
|---|---|---|---|---|---|
| | Protocol discriminator | Protocol discriminator 10.2 | M | V | 1/2 |
| | Transaction identifier | Transaction identifier 10.3.2 | M | V | 1/2– 3/2 |
| | Activate PDP context accept message identity | Message type 10.4 | M | V | 1 |
| | Negotiated LLC SAPI | LLC service access point identifier 10.5.6.9 | M | V | 1 |
| | Negotiated QoS | Quality of service 10.5.6.5 | M | LV | 13-21 |
| | Radio priority | Radio priority 10.5.7.2 | M | V | 1/2 |
| | Spare half octet | Spare half octet 10.5.1.8 | M | V | 1/2 |
| 2B | PDP address | Packet data protocol address 10.5.6.4 | O | TLV | 4-24 |
| 27 | Protocol configuration options | Protocol configuration options 10.5.6.3 | O | TLV | 3-253 |
| 34 | Packet Flow Identifier | Packet Flow Identifier 10.5.6.11 | O | TLV | 3 |
| 39 | SM cause | SM cause 2 10.5.6.6a | O | TLV | 3 |
| B- | Connectivity type | Connectivity type 10.5.6.19 | O | TV | 1 |
| C- | WLAN offload indication | WLAN offload acceptability 10.5.6.20 | O | TV | 1 |
| 33 | NBIFOM container | NBIFOM container 10.5.6.21 | O | TLV | 3 – 257 |

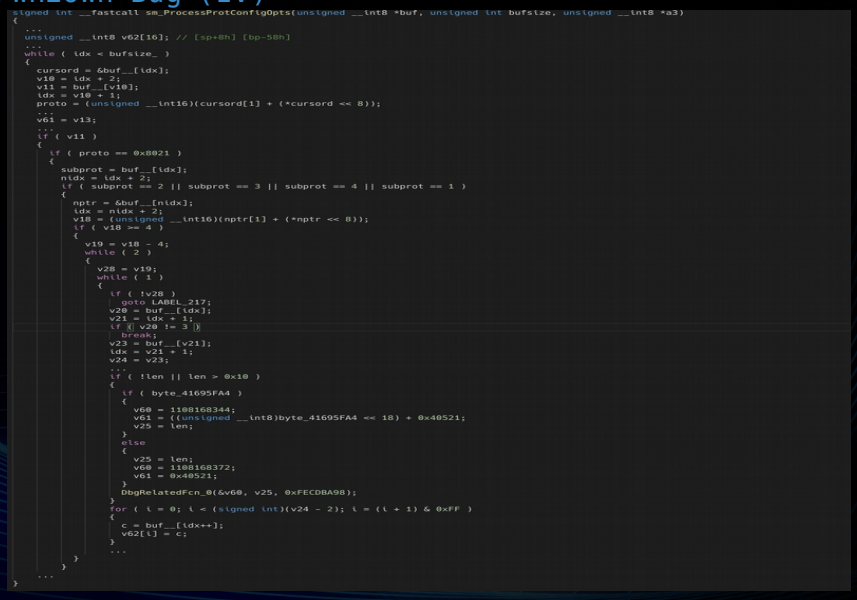
## The Mobile Pwn2Own Bug (III)

By reversing the SM task, we find the handlers for the different messages.

One of these messages is the ACTIVATE PDP CONTEXT ACCEPT message.

One part of it that seems to be interesting is the Protocol Configuration Options, the function processing that IE seems complicated.

## The Mobile Pwn2Own Bug (IV)



```
signed int __fastcall sm_ProcessProtConfigOpts(unsigned __int8 *buf, unsigned int bufsize, unsigned __int8 *a3)
{
  ...
  unsigned __int8 v62[16]; // [sp+8h] [bp-58h]
  ...
  while ( idx < bufsize_ )
  {
    cursord = &buf__[idx];
    v10 = idx + 2;
    v11 = buf__[v10];
    idx = v10 + 1;
    proto = (unsigned __int16)(cursord[1] + (*cursord << 8));
    ...
    v61 = v13;
    ...
    if ( v11 )
    {
      if ( proto == 0x8021 )
      {
        subprot = buf__[idx];
        nidx = idx + 2;
        if ( subprot == 2 || subprot == 3 || subprot == 4 || subprot == 1 )
        {
          nptr = &buf__[nidx];
          idx = nidx + 2;
          v18 = (unsigned __int16)(nptr[1] + (*nptr << 8));
          if ( v18 == 4 )
          {
            v19 = v18 - 4;
            while ( 2 )
            {
              v28 = v19;
              while ( 1 )
              {
                if ( !v28 )
                  goto LABEL_217;
                v20 = buf__[idx];
                v21 = idx + 1;
                if ( v20 != 3 )
                  break;
                v23 = buf__[v21];
                idx = v21 + 1;
                v24 = v23;
                ...
                if ( !len || len > 0x10 )
                {
                  if ( byte_41695FA4 )
                  {
                    v60 = 1108168344;
                    v61 = ((unsigned __int8)byte_41695FA4 << 18) + 0x40521;
                    v25 = len;
                  }
                  else
                  {
                    v25 = len;
                    v60 = 1108168372;
                    v61 = 0x40521;
                  }
                  DbgRelatedFcn_0(&v60, v25, 0xFECDBA98);
                }
                for ( i = 0; i < (signed int)(v24 - 2); i = (i + 1) & 0xFF )
                {
                  c = buf__[idx++];
                  v62[i] = c;
                }
                ...
              }
            }
          }
        }
      }
      ...
    }
  }
```

I am well aware that you can’t read this code, I don’t want you to read it.

It doesn’t really matter what this code is really doing but we know that if we control the contents of `buf` we can reach the vulnerable path

Turns out it is an IPCP (IP Control Protocol) header.

## The Mobile Pwn2Own Bug (V)

Processes Protocol Configuration Options which are sent by the Network in a `ACTIVATE PDP CONTEXT ACCEPT`.

PDP stands for Packet Data Protocol

The purpose of the protocol configuration options information element is to:

- transfer external network protocol options associated with a PDP context activation, and
- transfer additional (protocol) data (e.g. configuration parameters, error codes or messages/events) associated with an external protocol or an application.

Internet Protocol Control Protocol (IPCP) is a Network Control Protocol (NCP) for establishing and configuring Internet Protocol over a Point-to-Point Protocol link

## The Mobile Pwn2Own Bug (VI)

One of the supported protocols is **IPCP** (Internet Protocol Control Protocol).

**IPCP header:**

| *Offsets* | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Code | | | | | | | | Identifier | | | | | | | | Length | | | | | | | | | | | | | | | |

**Code.**

8 bits.
Specifies the function to be performed.

| Code | Description | References |
|---|---|---|
| 0 | Vendor Specific. | RFC 2153 |
| 1 | Configure-Request. | |
| 2 | Configure-Ack. | |
| 3 | Configure-Nak. | |
| 4 | Configure-Reject. | |
| 5 | Terminate-Request. | |
| 6 | Terminate-Ack. | |
| 7 | Code-Reject. | |

Internet Protocol Control Protocol (IPCP) is a Network Control Protocol (NCP) for establishing and configuring Internet Protocol over a Point-to-Point Protocol link

## The Mobile Pwn2Own Bug (VII)

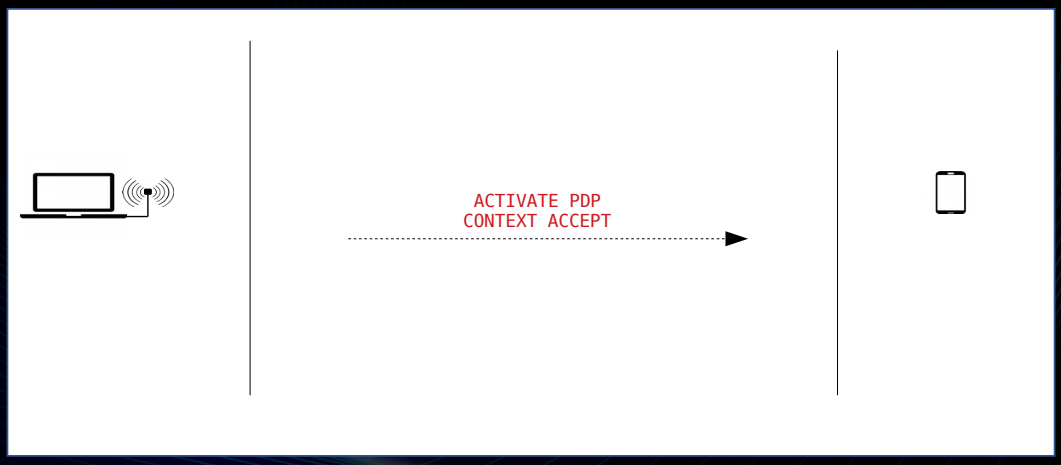
```
int sm_ProcessProtConfigOpts(unsigned __int8 *buf, unsigned int bufsize, unsigned __int8 *a3)
{
    unsigned __int8 tbuf[16];
    int idx;
    int len;
    unsigned __int8 c;

    ...

    len = buf[idx++];
    ...
    if ( !len || len > 0x10 )
    {
        // Do nothing about it
    }
    for ( i = 0; i < len - 2; i = (i + 1) & 0xFF )
    {
      c = buf[idx++];
      tbuf[i] = c;
    }

    ...
}
```

It doesn’t really matter what this code is really doing but we know that if we control the contents of `buf` we can reach the vulnerable path

Turns out it is an IPCP (IP Control Protocol) header.

## The Mobile Pwn2Own Bug (VIII)

The plan looks like this:




SM because we assume that there shouldn’t be much difference between GMM and MM.

## The Mobile Pwn2Own Bug (IX)

Not so easy...

Problem is the phone will only process this message if it is in the correct state.

This happens when the phone sends a `ACTIVATE PDP CONTEXT REQUEST` message.

Which in turn happens if the phone is manually configured to include an APN in the connection settings.

However this is a problem for the P2O...

## The Mobile Pwn2Own Bug (X)

Read more of the technical standards...

We can force the MS get in the correct state (i.e perform PDP activation procedure) by sending a `REQUEST PDP CONTEXT ACTIVATION`.

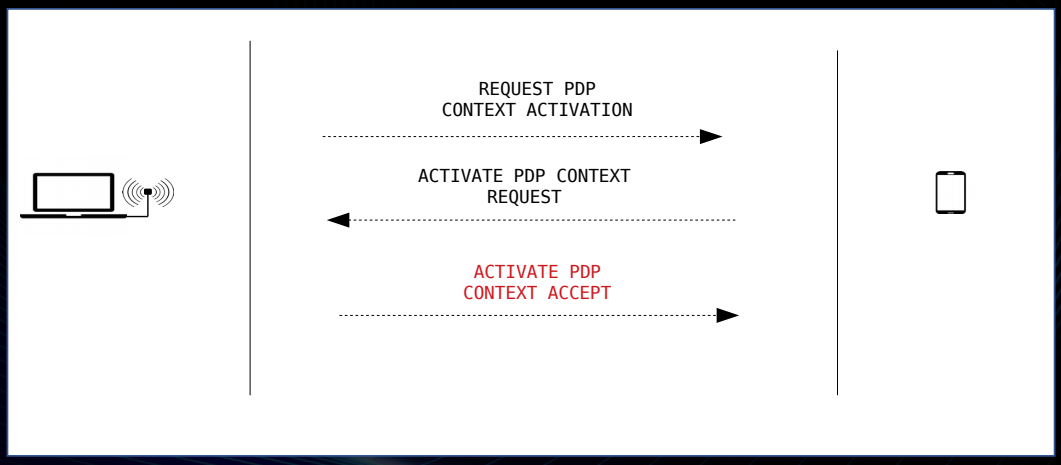
### 9.5.7 Request PDP context activation

This message is sent by the network to the MS to initiate activation of a PDP context.
See table 9.5.7/3GPP TS 24.008.

| | |
|---|---|
| Message type: | REQUEST PDP CONTEXT ACTIVATION |
| Significance: | global |
| Direction: | network to MS |

## The Mobile Pwn2Own Bug (XI)

The actual plan looks like this:



## The Mobile Pwn2Own Bug (XII)

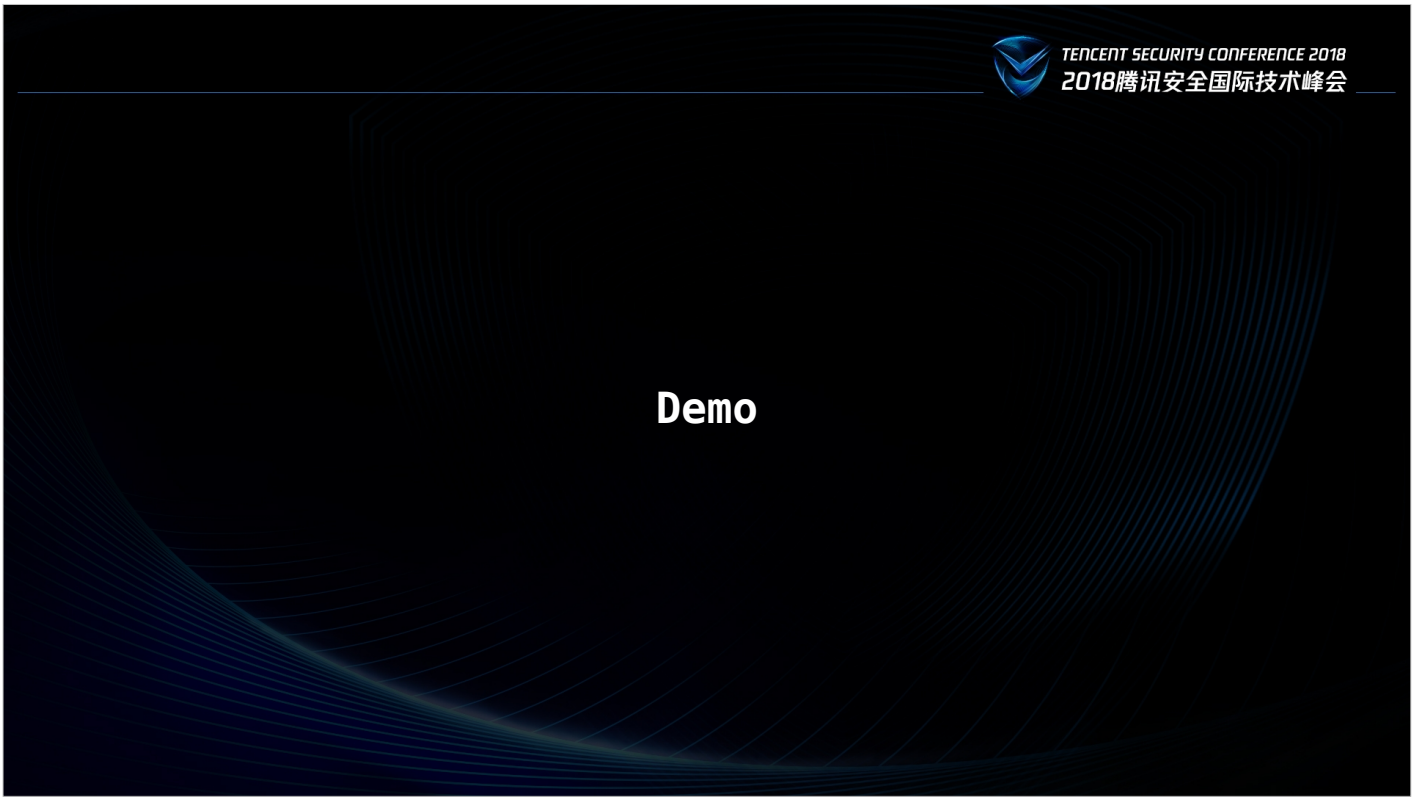
ROP is needed for the first stage of your payload due to ARM cache-fu.

Copy shellcode to some arbitrary RWX address and invalidate/flush the i-cache/d-cache.

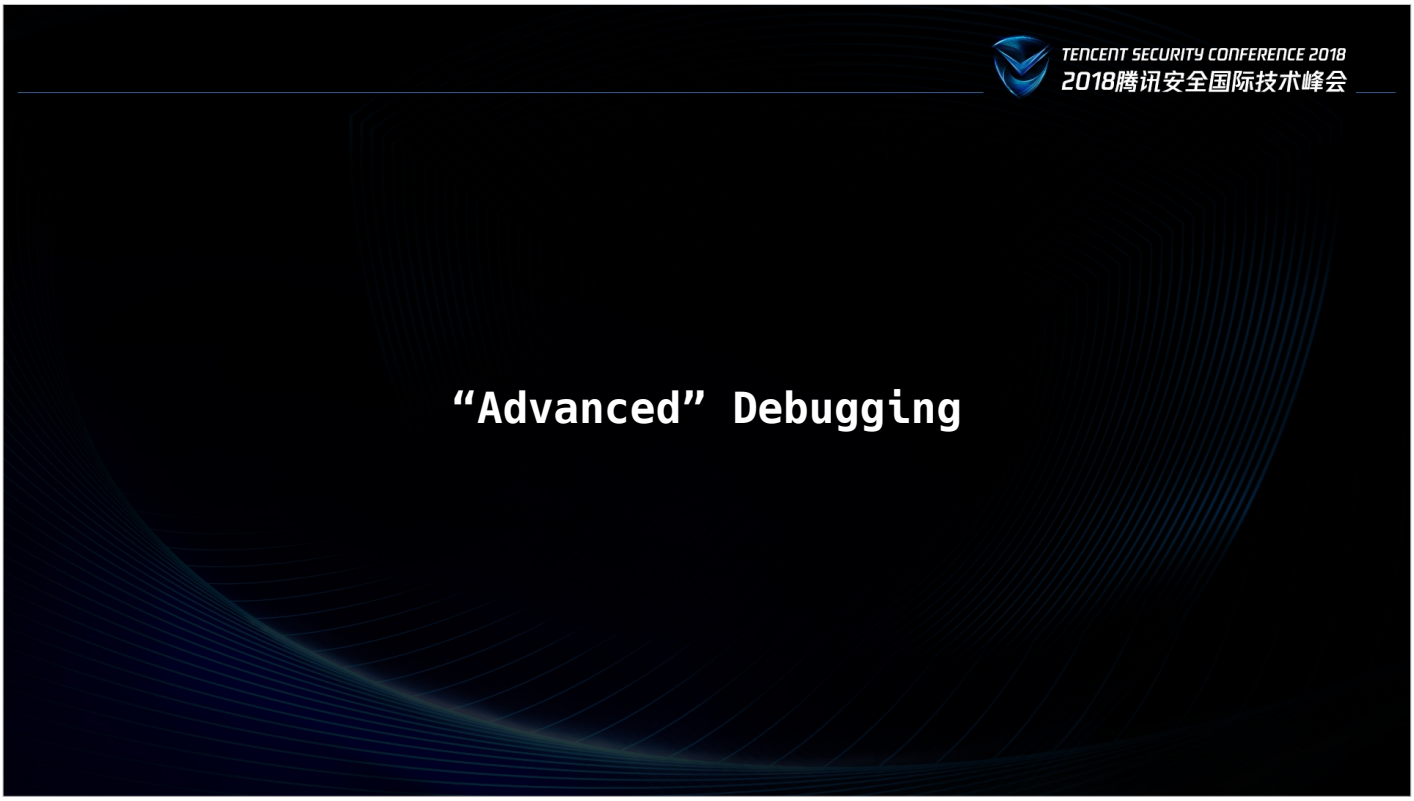
Jump to win.

Payload can do any number of things, for P2O I chose to write to the Android filesystem by leveraging the **RFS** (Remote? File System), a mechanism which allows the baseband to store data such as NV Items to the android filesystem.

Payload can even be a custom “debugger” that can be used to find other bugs and write more involved exploits (e.g heap memory corruption).




Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.




Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

## Custom Debugger (I)

As mentioned previously, we can use the shellcode as a debugger.

Modify .text to insert “breakpoints”.

```
1   void set_bp(void * addr, int t){
2       unsigned int * addr32 = (unsigned int *)addr;
3       unsigned short * addr16 = (unsigned short *)addr;
4       if(t){
5           addr32[0] = 0xf04f;
6           addr16[1] = 0x37b1;
7           addr16[2] = 0x603f;
8       }else{
9           addr32[0] = 0xe51fc000;
10          addr32[1] = 0xe58cc000;
11          addr32[2] = 0xb1b1b1b1;
12      }
13      flush_dache_range(addr, 0x100);
14      invalidate_icache_brpred();
15  }
```

## Custom Debugger (II)

Hook *malloc()* and *free()* and log to some file using *RFS*. Get trace from resulting crashdump.

```
1  void heap_tracer(){
2      // patch malloc() and free()
3      unsigned int ** addr = (unsigned int **)MALLOC_FUNC_PTR;
4      *addr = (unsigned int *)&hmalloc;
5      addr = (unsigned int **)FREE_FUNC_PTR;
6      *addr = (unsigned int *)&hfree;
7  }
8  void * hmalloc(int pool, int size, char * file, int line){
9      char tmpbuf[256];
10     unsigned int * res = 0;
11     res = malloc(pool, size, file, line);
12     snprintf(tmpbuf, sizeof(tmpbuf), "malloc(%d, %d, %s, %d) = 0x%p\n", pool, size, file, line, res)
13     log_msg(tmpbuf);
14     return res;
15 }
16 void hfree(void * paddr, char * file, int line){
17     char tmpbuf[256];
18     if(paddr){
19         unsigned int * addr = *((unsigned int **)paddr);
20         snprintf(tmpbuf, sizeof(tmpbuf), "free(*0x%p = 0x%p, %s, %d)\n", paddr, addr, file, line);
21         log_msg(tmpbuf);
22     }
23     free(paddr, file, line);
24 }
```

## Custom Debugger (III)

*free()* is called with: pointer to pointer, calling file and line number.

```
free(*0x41691278 = 0x437a05bc, ../../../HEDGE/GL3/GRR/Code/Src/rr_main.c, 1713)
free(*0x42efab98 = 0x43765820, ../../../HEDGE/NASL3/MM/Code/Src/mm_Utilities.c, 506)
free(*0x41691278 = 0x437a05a0, ../../../HEDGE/GL3/GRR/Code/Src/rr_main.c, 1713)
free(*0x41693eb0 = 0x437a023c, ../../../HEDGE/NASL3/MM/Code/Src/mm_Main.c, 609)
free(*0x4296d1b0 = 0x437ac03c, ../../../HEDGE/GL2/LLC/Code/Src/llc_Data.c, 5411)
free(*0x43010bb8 = 0x437a00bc, ../../../HEDGE/GL2/LLC/Code/Src/llc_Main.c, 173)
```

*malloc()* is called with: “pool” number, size, calling file and line number.

```
malloc(4, 13, e/Src/ns_OsInterface.c, 160) = 0x43799360
malloc(4, 13, e/Src/ns_MsgHandler.c, 321) = 0x43799460
malloc(4, 13, e/Src/ns_MsgHandler.c, 321) = 0x43799520
malloc(4, 4, ../../../HIU/MCD/HIC/SIPC/Code/Src/sipcCallCmdHandle.c, 3079) = 0x4376e5e0
malloc(4, 20, ../../../HIU/COMMON/HID/HostIF/Code/Src/hostifTransportMgr.c, 1978) = 0x43799360
malloc(4, 122, ../../../HIU/MCD/HIC/SIPC/Code/Src/sipcSSCmdhandle.c, 2231) = 0x43797420
```

## Custom Debugger (IV)

We can also use the RFS functionality mentioned above to build a debug server.

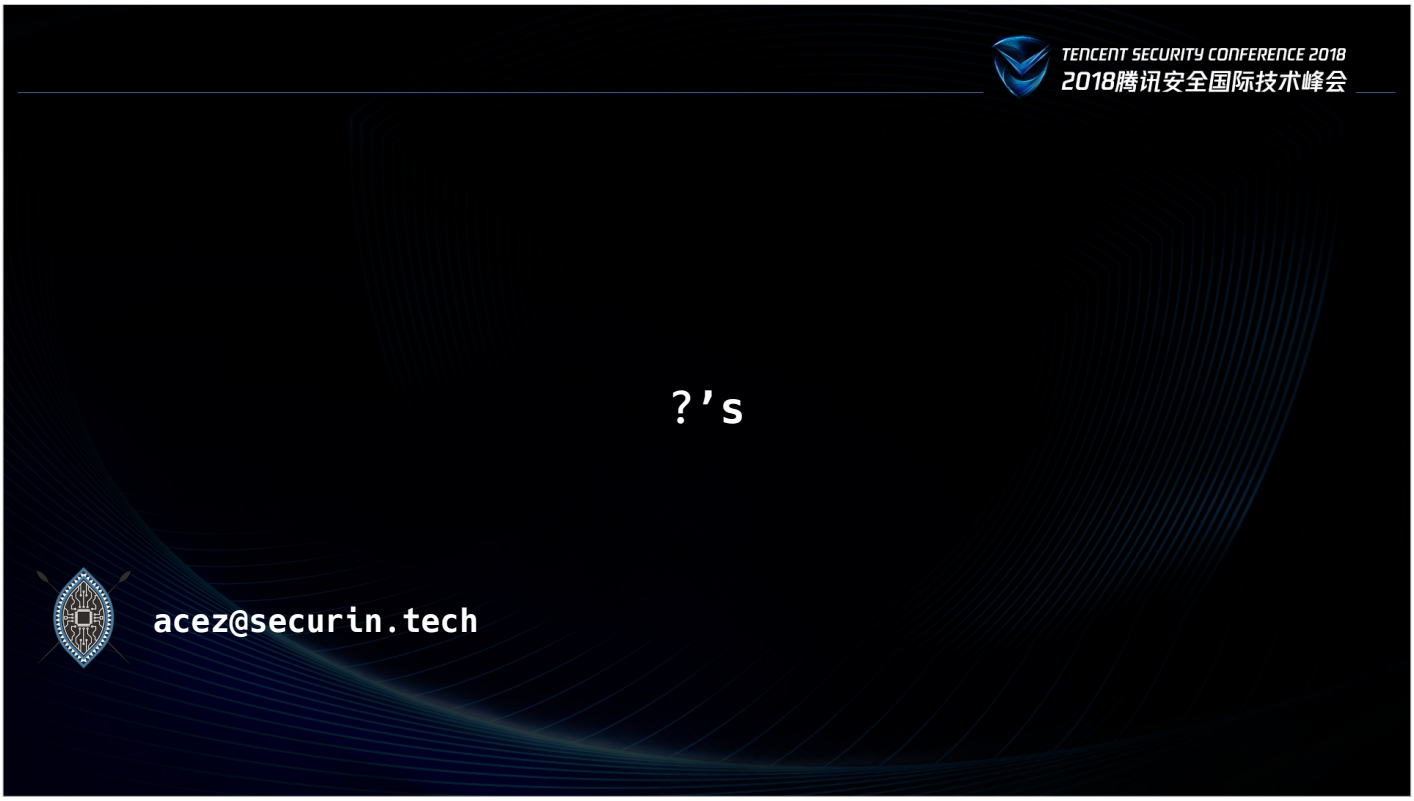
The client on the Application Processor and server on the Cellular Processor communicate through a file on the Android filesystem.

Can be used to implement basic functionality such as live memory inspection, breakpoint setting, code patching, etc.

Can also be used to debug the interface between CP and AP by adding functionality to send arbitrary messages through the IPC channels.


Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.

- Baseband exploitation isn’t as hard as it is percieved to be.
- You don’t need to know much about cellular networks in order to exploit them.
- The cellular protocol stacks are complicated enough that there should be many bugs still waiting to be discovered.
- Many targets out there, Mediatek, Huawei, Intel, Qualcomm...




Here, we will give a little bit of background on what cellular networks are, describe the baseband, how the two communicate. An intro to the different technologies and protocols that are of interest to us as security researchers. Basically, a bunch of things that we need to know that will help us in playing with this kind of research.