uc3m | Universidad Carlos III de Madrid

Bachelor's Degree in Computer Science and Engineering
2020-2021

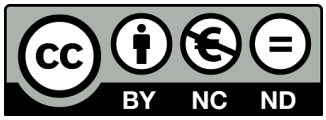
Bachelor Thesis

"Understanding Mobile Phones Baseband Vulnerabilities: From Exploit to Warehouse"

Adrián Hacar Sobrino

Antonio Nappa

Leganés, July 2021



Anything out there is vulnerable to attack given enough time and resources.

Kevin Mitnick

ABSTRACT

During the last years we have experienced a transition from desktop computers to mobile devices, they are much more versatile, we can carry them in our pocket for the whole day and they incorporate a lot of technologies and functionalities, such as GPS, Bluetooth, NFC, Radio, etc. Among them, one of the most powerful is that we have the ability to make calls and surf the internet using cellular data connection wirelessly just by purchasing the services of a mobile company. For this to work, the phone needs to communicate with the cellular antennas of the service provider using a device embedded in all modern phones, the baseband processor. The baseband processor sends and receives large amounts of data to implement the complex protocols that are used for everything to function well. A programming bug in the implementation of these protocols can lead to a security vulnerability, putting at risk all the information that is carried in the phone.

In this work is presented research about baseband processors from a security point of view, making emphasis on memory corruption vulnerabilities. The operating system running in this baseband processor is studied applying reversing engineering techniques to find n-day vulnerabilities, these types of vulnerabilities have been already reported but there are still devices that have not been patched. It is estimated that 8.66 million devices were sold as brand new in 2020 containing one of the known vulnerabilities identified in this work. Furthermore, an over-the-air attack is carried out in a controlled environment using a real phone and a Software defined Radio simulating a phone carrier antenna.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

This work is an investigation to measure the impact and how much we are exposed to a type of vulnerability called n-day. These are already known and fixed by the manufacturer, however there are plenty of devices that do not get the security patches. A lot of them are discontinued and no longer maintained by the company and others are brand new phones that are stored in a warehouse waiting to be purchased at the moment the vulnerability is discovered. Therefore a user would be buying a brand new phone with a known vulnerability that can be exploited.

Section 5 describes a statistical estimation of how many brand new Samsung devices were sold in 2020 exposed to a vulnerability in Samsung's baseband processor. This statistical study serves as the main motivation for this work.

Some years ago, in order to perform this investigation, thousands of euros were needed to set up a lab environment. Nowadays, with the existence of Software-Defined-Radios (SDR), research on cellular communications has become much more affordable.

The question that this work is going to answer is: **is possible to locate and exploit an n-day baseband vulnerability using publicly available resources and affordable lab equipment?**

Note that this work is developed with the purpose of finding the vulnerabilities so it can be fixed as soon as possible and users are aware of how much we are exposed. Contributing to public security research helps companies to fix bugs and develop safer products for our society. It is highly recommended to check Section 4 to learn the legal aspects that are involved in the development of this research.

1.2. Objectives

The main objectives of this work are briefly described down below:

- Understand how Mobile communications protocols work and its security concerns.
- Study how Samsung's baseband firmware works using reversing engineering techniques.
- Find and exploit vulnerabilities in Samsung's baseband processor crashing the device.
- Measure how many devices are exposed to n-day vulnerabilities and the impact of them.

1.3. Document structure

- Chapter 1, *Introduction*, contains the motivation and the objectives of this work.
- Chapter 2, *State of the art*, contains the concepts needed to understand the following work and the prior research carried out regarding baseband processors.
- Chapter 3, *Approach*, shows the process carried out to find and exploit vulnerabilities in the Samsung baseband.
- Chapter 4, *Legal Framework*, contains the regulations that need to be considered for the development of this work.
- Chapter 5, *Socio-Economic Framework*, review the motivation of this work considering the socio-economic impact.
- Chapter 6, *Budget*, describes the resources used in this work.
- Chapter 7, *Conclusions and results*, review the objectives of this work, the results obtained and my personal conclusion.

2. STATE OF THE ART

This chapter shows the concepts that are needed in order to understand the research developed in this work. A general introduction about the main components and technologies that are involved in the functioning of a phone is provided in Section 2.1. Then, an overview of Cellular Communications protocols is given in Section 2.2. Next, the device that implements these cellular protocols within a phone, i.e, the baseband processor, is presented in Section 2.3. The following Section 2.4 provides an overview of general concepts about computer security as well as an explanation of memory corruption vulnerabilities that will be of interest in order to understand the work carried out in Section 3.

Section 2.5 presents the related work regarding the research in baseband processors that will serve as a baseline for the investigation carried out in Section 3.

2.1. How phones work

A mobile phone is a complex system that contains inside many processors in charge of multiple tasks. The main one is the AP (Application Processor) that provides all system capabilities needed to support device's applications.

Nevertheless, a phone carries many other devices that perform different tasks, like the WIFI, NFC, Bluetooth radio, etc. In this work we are going to focus on another component, called the *baseband processor*. This component is the actual "phone part" of a phone, in charge of managing the cellular radio functions. In other words, this is the part of the phone that communicates with the cellular network and allows us to make calls, send text messages and surf the internet using data connection. In Section 2.3, what a baseband processor is and how it works is explained in detail.

The terms baseband, communication processor(CP) and modem will be used indistinctly to refer to the same component of the phone.

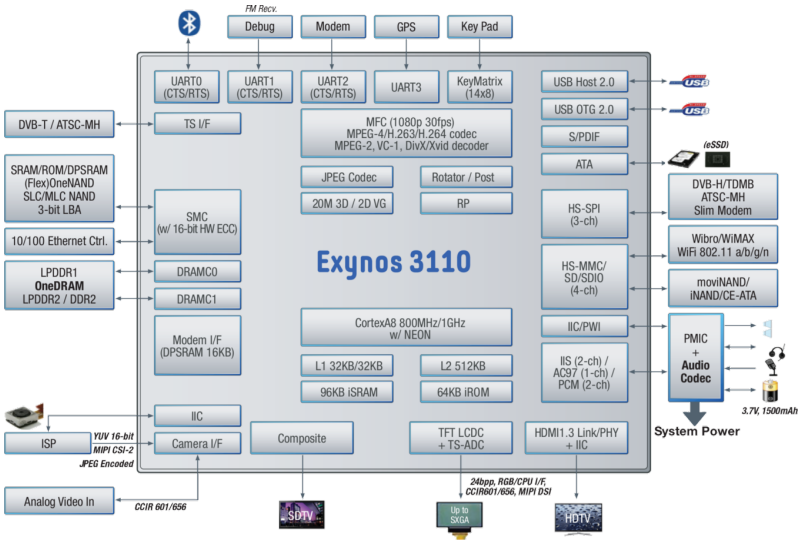


Fig. 2.1. System-on-Chip diagram example. Exynos 3110

Most of the elements mentioned above, usually are integrated in a SoC (System on Chip). See Figure 2.1. Nevertheless, it is very important to highlight that the baseband firmware runs on a dedicated chip and a **different operating system** to the one running on the Application Processor(typically Android or iOS).

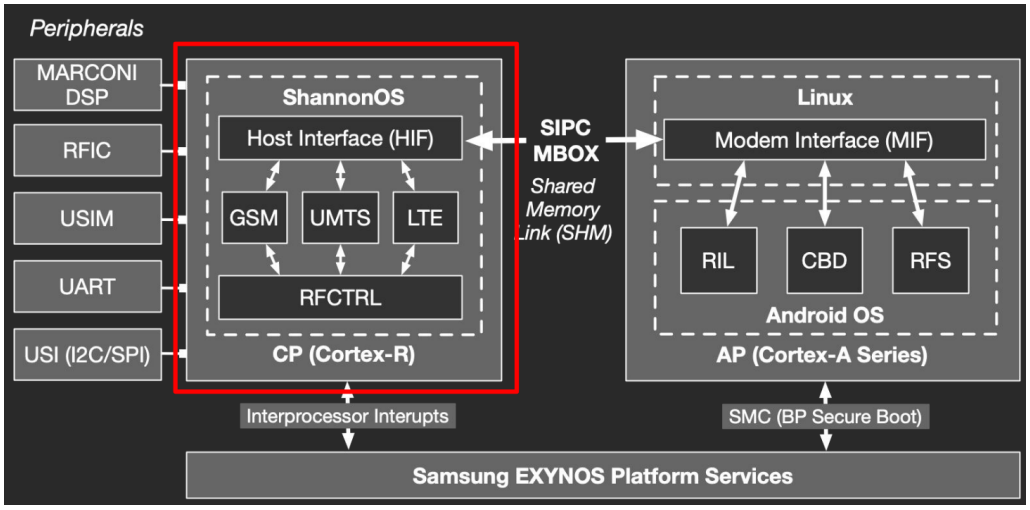


Fig. 2.2. Application Processor and Baseband Layout [1]

Figure 2.2 shows the baseband and the Application processor architecture within an Exynos SoC. At the left, there is the baseband processor that implements the cellular protocols(GSM, UMTS, LTE). This will be explained in detail in Section 2.3.1.

At the right side, there is the AP(Application Processor), running the Operating System, Android, over the Linux kernel. This operating system includes programs and services to interact with CP(Communication Processor):

- **RILD:** The Radio Interface Layer Daemon is a program running on the AP that

provides an interface with the modem.

- **CBD:** CP Boot Daemon loads the firmware image in the baseband processor.
- **RFS:** Remote File System stores the modem configuration and provides access to the Android filesystem from the modem.

2.2. Cellular Communications

When a text message is sent or a phone call is made, the device needs to communicate with the cellular network using a set of protocols. These are defined using a standard. For example the *Global System for Mobile Communications* (GSM) describes the second-generation(2G) standard for cellular communications[2]. In this section I will present an overview of how these protocols work, the security concerns of the first generation and how the following generations solved these problems.

2.2.1. GSM and GPRS

The Global System for Mobile Communications (GSM) is a standard developed by the European Telecommunications Standards Institute (ETSI) that provides a mobile communication service being the second generation of digital cellular networks(2G).

In order to use this service, the subscriber needs a mobile equipment(ME) and a Subscriber Identity Module (SIM card). The ME is the device used to connect to the network, in short, the mobile phone itself. The SIM card contains data used to identify and authenticate the user in the network. The most important information contained in this SIM card is the *International Mobile Subscriber Identity* (IMSI), this number is unique and is used to identify a subscriber universally. A private key *Ku* and a set of ciphering algorithms(A3, A5, A8) are also stored in the sSIM card in order to authenticate and encrypt the data transmitted. See Figure 2.3

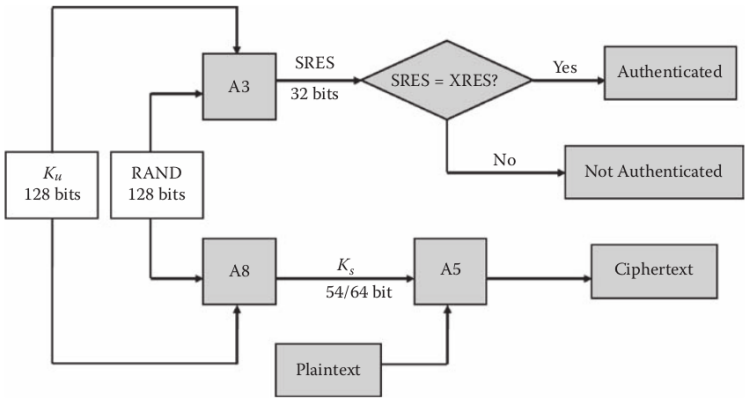


Fig. 2.3. GSM Cryptographic model [3]

Both the SIM card and the network knows a secret key, *Ku*. When the user connects to the network, the network sends a random number, *RAND*. Using the secret key, *Ku*, and the *RAND* along with the algorithm *A3* another value is computed by the ME and the network. These values are *SRES* and *XRES* computed by the ME and the network respectively. The network checks if these values match, if they are the same means that the ME has the correct *Ku*, therefore the SIM card is authenticated. [3]

Using the secret key and the previously mentioned random value, a session key, *Ks* is created using the algorithm *A8*. This session key will be used to encrypt all the data using the *A5* algorithm.

One of the main problems of this security model is that there is no network authentication. This means that the user is authenticated against the network, but this does not have a way to know if the network it is connecting to is legitimate.

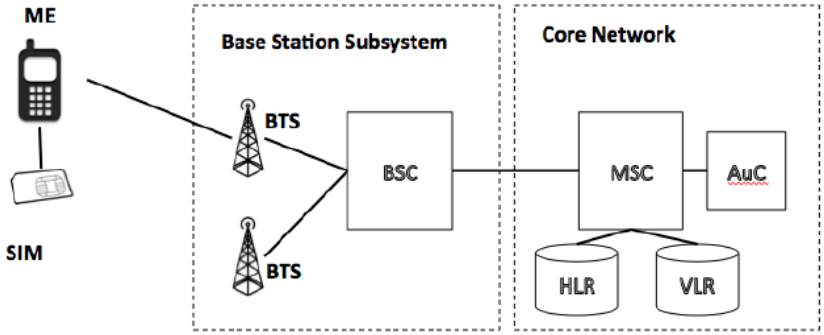


Fig. 2.4. GSM structure diagram[4]

In the Figure 2.4 a schema is presented with the most important elements of the GSM infrastructure. Two parts are differentiated. First, there is the *Base Station Subsystem* (BSS) that is in charge of communicating with the mobile phone in the network. Second, there is the *Network Switching Subsystem* (NSS), also known as *Core Network* that is composed of several elements that switch the calls, authenticates and tracks the users among others. Below, a brief description of the main elements that compose the GSM network is given.

- **ME:** The Mobile equipment communicates directly with the Base Transceiver Station (BTS). Each one is part of a set of stations controlled by a Base Station Controller (BSC).
- **BSC:** The Base Station Controller is in charge of managing hangover, radio-channel setup and frequency hopping among others.

- **MSC:** The Mobile Switching Center is the main component of the GSM network. It is responsible for creating an end-to-end connection with the other ME. It works as a switching node and provides all the management functions such as authentication, location and handover. It is also in charge of routing all incoming and outgoing calls.

- **HLR:** The Home Location Register is a database that contains the IMSI of the subscriber along with all the subscriber parameters, including the shared authentication key *Ku*.

- **VLR:** The Visitor Location Register contains subscriber information as the HLR, but the VLR is assigned to a specific area. This way, information is replicated in order to speed up and to not to load in excess the HLR.

- **AuC:** The Authentication Center is in charge of authenticating each SIM card and generating the key used to encrypt all the data sent between the Mobile Equipment and the network.

One of the main problems of GSM is that as mentioned earlier, the user authenticates itself against the network, but **the network is not authenticated by the user**. This means that an attacker can operate a False BTS and make the victim connect to it. Then, the attacker will create a new connection with the real BTS from the service provider. This way an attacker can sniff all the traffic. The technique described above is known as a *Man In The Middle Attack*

Another possible scenario is that the attacker sends a crafted package to the Mobile Station that triggers a vulnerability in the baseband processor(See Section 2.3). This scenario is replicated with a Software Defined Radio emulating a BTS and a physical phone in Section 3.2.

2.2.2. Next generations

The next generations of wireless mobile telecommunications mean not just an improvement in the speed of the data transmission. It also means a significant advance in security.

Among other security characteristics and mechanisms, the third generation of wireless mobile telecommunications, UMTS, incorporates a major update, the mutual authentication. Allowing the User Equipment to be sure that the network is connecting to is legitimate and not a fake one. This mechanism is known as the *UMTS authentication and key agreement (UMTS AKA)*. The main idea relies on the fact that both parties need to prove to each other that they know the shared secret without revealing it.

Note that in order to perform dynamic research (See Section 3.2) over these protocols

where mutual authentication exists a programmable SIM card is needed, otherwise, the ones linked to a phone provider will refuse to connect to our Base Station, since the secret in the SIM card is not accessible.

2.3. Baseband Processor

In this section an explanation about what a baseband processor is and how it works is given. Then, we will deepen in a particular baseband processor developed by Samsung called *Shannon*. The concepts explained in this section are needed in order to understand prior research and the approach carried out in order to find vulnerabilities in *Shannon*. (See Sections 2.5 and 3)

As introduced in Section 2.1, the baseband processor is the part of the phone that is in charge of managing the communications with the cellular network, being this another separate chip within the mobile device apart from the Application Processor. Specifically the baseband processor is in charge of the first three layers of the OSI model (L1: Physical layer, L2: Data Link layer and L3: Network layer). The upper layers are implemented in the AP.

A use case to receive an SMS will happen as follows: A SMS is received over the network, first, it gets handled by the CP. This will place the relevant information of the packet in a shared memory region. Then, a daemon running in the AP(RILD) will decode the message and send it to the relevant process in the AP. In this case, the Messaging application presents the SMS message to the user.

Each manufacturer is in charge of implementing the standards [2] to handle the communication with the mobile network. In the Figure 2.5 we can see the most important manufactures of baseband processors. *Shannon* is the firmware implementation of Samsung.

Note that at the time of writing this work, even though Samsung builds their own System-on-chip (Exynos), the devices that are sold in the United States carry Qualcomm chips. The reason is that in the US, in contrast to GSM, there is another standard called *IS-95* or *CDMA2000*(Third generation). This standard was developed by Qualcomm, company that holds the CDMA patents. Therefore, Samsung devices sold in the US carry the Qualcomm System-on-Chip, hence, **not *Shannon*** but Qualcomm baseband.

Fig. 2.5. Baseband processor manufacturers

2.3.1. Hardware

The Samsung baseband firmware, Shannon, is running on an *ARM Cortex R7*. These type of processors are optimized for hard real-time applications where very low latency is required. Huawei and Mediatek basebands also run on a *ARM Cortex-R7* as well as many other embedded systems that deals with communication, such as *Bluetooth*, *NFC* and *WIFI* (some of them run in an ARM Cortex-R3 depending on the required performance). Since all the baseband code is designed to run over this hardware, it is very important to have an overview of how this ARM hardware implementation works. Some of the most relevant characteristics of this microprocessor are explained below [5][6]:

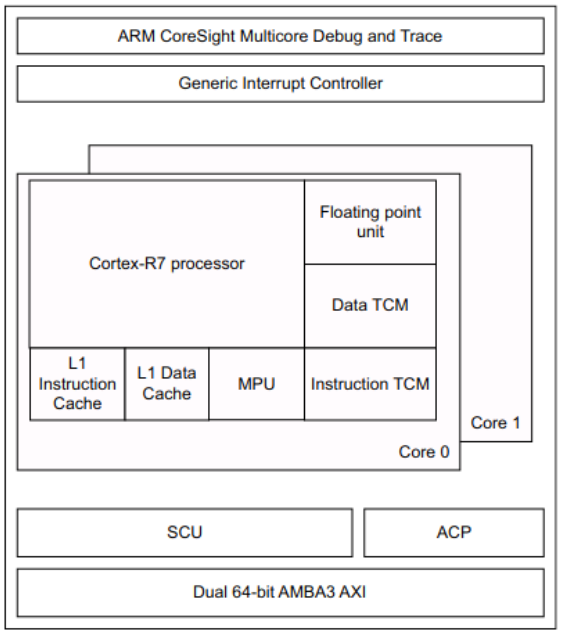


Fig. 2.6. Cortex-R7 Processor Block diagram [6]

- **It has a 11-stage pipeline with instruction prefetch, and out-of-order pipeline:** This means that the execution of an instruction is divided in 11 stages. Therefore, many instructions can be run at the same time since one instruction can be in a different stage of the pipeline. For example, one instruction can be in the Fetch state while another one is in the Decode or Execution stage. As it uses pre-fetch,

some instructions can be loaded from Main memory into a cache memory before they are needed in order to speed up the processor.

- **Dynamic execution:** Out of order execution is used to avoid some stalls in the pipeline, this is achieved by executing some instructions in a different order instead of running them always sequentially. The hardware takes care of all the possible hazards, this means that even when some instructions are not being executed in the original order, the output is always the same, even when there are dependencies between the instructions.

- **Register renaming:** This technique is used to avoid false data dependencies, specifically output dependencies and antidependencies. The processor transposes the logical registers to physical registers. This technique can be exploited by the out-of-order execution characteristic of this processor.

- **Static and dynamic branch prediction:** Static branch prediction is done in the decoding stage without taking into account the history of other branch decisions. In the case of this processor, it speculatively takes backward branches rather than forward branches. For this, the processor detects if the target address is lower than the current address. In case it is lower, it takes the branch, this is very accurate when executing a code with a lot of loops. In contrast, the dynamic branch prediction makes use of historical information about whether conditional branches are taken or not taken on previous executions. In the Cortex-R7 a *Branch Target Address Cache* (BTAC) is used to hold information about the previous branch instruction execution. By default the processor uses dynamic branch prediction, it is in the case where there is no history when static branch prediction is used.

- **Cache Memory and TCM** As we can see In the Figure 2.6, there are two cache memories L1, for instructions and data. This cache memory sits between the core and main memory and it is significantly faster than main memory. In addition, this ARM processor has another memory called *Tightly Coupled Memory* (TCM). This memory has also very low latency and can be used by the developer to hold time-critical routines or data, such as interruption handlers. This way, the processor has immediate access to the subroutine instead of fetching the instructions from main memory. The most important difference between cache memory and TCM, is that when using cache memory the data or instruction requested might not be on the cache (cache miss) requiring a slower access to memory. In contrast, TCM is used as part of the physical memory map and does not need to be associated with another physical address. It is accessed the same way as main memory but much faster. This provides a fast and consistent way to store and read critical data.

- **FPU**: The Cortex-R7 counts with a Floating Point Unit with 16 double-precision registers or 32 single-precision registers. This unit supports add, substract, divide, multiply, accumulate and square root operations.

- **MPU**: The Memory Protection Unit is a component that controls access to memory. The memory is splitted in regions, each of them programmed with a base address and size. Each region has its individual properties, like the permissions: Writable, Readable and Executable. These are programmed using the instructions: *MRC* and *MCR*. As explained, the TCM is also part of the memory map, therefore the MPU must be programmed to cover the complete TCM address space. This is achieved using specific registers. Note that if an attacker achieves code execution in the baseband, these instructions can be used to reprogram the MPU, changing the permissions of regions of interest. The special registers used to configure the MPU are called coprocessors and are also used to configure other critical operations, like booting. In order to configure the MPU the coprocessor *p15* needs to be modified. Figure 2.7 shows the function that reprograms the MPU in Samsung's baseband processor.

```
; void __cdecl program_mpu(int drbar, int drsr, int dracr, int rgnr)
program_mpu                             ; CODE XREF: calls_program_mpu+40↓j
                MCR             p15, 0, R3,c6,c2, 0 ; Write MPU Region Number Register
                MCR             p15, 0, R0,c6,c1, 0 ; Write MPU Region Base Address Register
                MCR             p15, 0, R1,c6,c1, 2 ; Write MPU Region Size and Enable Register
                MCR             p15, 0, R2,c6,c1, 4 ; Write MPU Region Access Control Register
                BX              LR
; End of function program_mpu
```

Fig. 2.7. MPU configuration

The Cortex-R7 in a Samsung phone has its own RAM and its own operating system but it shares data with the Application Processor through communications interfaces using shared memory and a serial link (See Figure 2.8). Even though the baseband and the AP are completely different processors, since there is a communication interface, if an attacker gets execution code in the CP, it is possible to do an escalation to the AP. Section 2.1 explains how this communication between the Application processor and the baseband happens.

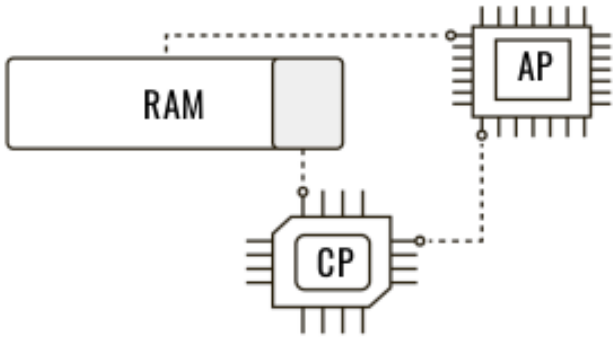


Fig. 2.8. CP-Cellular processor and AP-Application Processor diagram

2.3.2. Operating System

Android[7] is the Operating System that is running on the AP. This is based on a modified version of the Linux Kernel[8], where the code is open source and well documented. Any researcher has access to the source code and can study it to find vulnerabilities.

In contrast, inside the ARM Cortex R7 is running the operating system *Shannon*. This software is fully composed of proprietary code, which means that the source code is not available to the public, neither there is documentation and there is little information online on how this operating system works. All the information from this software is obtained by researchers reversing engineering the compiled code. In section 3 I will show how to do this process of reverse engineering the software from scratch, getting the binary firmware of the baseband from publicly available sources and loading it into a disassembler.

However, there are concepts and procedures that are common to all basebands or have been already reported by other researchers and serve as a baseline to find vulnerabilities in this firmware. The main characteristics are explained below.

It works as a RTOS

The Shannon baseband works as a RTOS(Real Time Operating System), this means that the response to the events is guaranteed within a specified time constraint. In this RTOS, there is a region known as the PAL(Platform Abstraction Layer). Here, low level tasks are implemented, like task management, task synchronization and memory related functions. Nearly, a mix of a *kernel* and a *libc* equivalent in Linux.

In a RTOS, the equivalent of processes in a mainstream Operating System are called Tasks. Each task is independent from the other ones running on the operating system, although they communicate to each other with use of queues.

Each task has its own region in memory with its own code and stack. Shannon counts with a task scheduler as in any other operating system. In this case, the tasks are executed depending on its priority. Tasks that handle interrupts, such as messages received from the network are assigned a higher priority than maintenance tasks. In addition, each task has a state machine. This is needed in order to implement communication protocols. In the simplest scenario, an incoming message will be processed just when another message has been sent previously by the baseband. This means that the baseband needs first to be in the correct state before processing the incoming package. Section 2.5.2 describes a vulnerability that in order to be exploited the existence of these state machines needs to be taken into account.

All code is executed in kernel mode

Some operating systems like desktop ones have two kinds of execution modes. *Kernel Mode* and *User mode*. In the first case, programs have access to privileged instructions and complete access to the hardware. However, the programs that run in User Mode do not have permission to run privileged instructions and need to use API's provided by the system in order to execute privileged tasks.

In contrast, Shannon is implemented as a RTOS where all the code is running in Kernel Mode. This means that if an attacker gets code execution in the baseband, this code will have the maximum privileges in the baseband processor. Note that this code execution would be running in the baseband processor, that is a different system than the Application Processor, therefore the attacker wouldn't have access to the application data, but other vulnerabilities can be exploited to access the AP from the baseband. The only layer protection is memory protection, but as I mentioned the MPU can be configured using the instructions *MRC* and *MCR*. This means that once an attacker achieves code execution, the MPU can be reprogrammed and change the permissions of the sections in memory. For example, an executable but not writable section of memory can be modified to be writable as explained in Section 2.3.1

Lack of basic protection techniques

In addition to that all the code is executed in kernel mode, Shannon does not have some of the basic protection mechanisms easing the work of an attacker. The two most important protection mechanisms that are not included in Shannon are the following:

- **DEP**: Data Execution Prevention is a mechanism to mark some memory regions as non-executable. This way if an attacker controls a return address, via a buffer overflow will not be able to write the code in memory to execute it. These areas are the Stack and the Heap, due to they need to be writable and readable but not executable.
- **ASLR**: Address Space Layout Randomization is a technique that randomize the position of the base of the executable in order to prevent that an attacker exploiting a memory corruption vulnerability jumps to a specific function or instruction in the firmware like the *MCR* and *MRC* already mentioned, that can be used to reconfigure the permissions of the memory sections.

Firmware is not encrypted but signed

Shannon's main code was encrypted in the Samsung Galaxy S6. But as Nico Gold and Daniel Komaromy explained in a presentation in the REcon it is possible to get the code from memory(See Section 2.5.1). In any case, from the *Samsung Galaxy S7* the firmware is not encrypted, therefore it is enough downloading the firmware from a public source[9]

and loading it into our debugger.

In contrast, all the baseband code running needs to be signed by Samsung, this means that it is not possible to modify Shannon original code directly and load it on a Samsung device. However, it is possible to reconfigure the MPU and overwrite the code section of the firmware.

2.4. Computer Security

This section presents some concepts related to computer security as risk, threat, software bug and vulnerability. Then, Section 2.4.1 deeply explains memory corruption vulnerabilities, needed to understand the vulnerabilities found in Section 3.

- **Threat:** Potential violation of Information Security.
- **Risk:** Probability that a threat materializes, exploiting a vulnerability and causing harm in a process or system.
- **Vulnerability:** Weakness of an Information System that can be exploited by a threat actor.
- **Software bug:** Error, fault or flaw in a computer system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

The concepts of Software Bug and vulnerability are often used interchangeably. However, a programming bug does not necessarily lead to a vulnerability.

2.4.1. Memory corruption vulnerabilities

Memory corruption vulnerabilities modify some parts of memory that were not intended to be modified. In programming languages where the memory is managed directly by the developer, this type of vulnerability is typically due to a programming mistake.

The most common type of memory corruption vulnerabilities are buffer overflows. This occurs when data is written into a memory region and the data transferred is bigger than the buffer allocated.

Fig. 2.9. Buffer overflow example

Consider Figure 2.9 and the Listing 2.1. A buffer with a length of 8 is allocated, but the amount of bytes copied to that buffer is 10, since the *strcpy()* function copies the data without checking the length, a buffer overflow is triggered.

When performing buffer overflow attacks, usually the objective is not to overwrite other variables, but to overwrite the return address to point to another memory location which code will be executed. In normal cases, the return address is used to resume the execution after the function has ended. (See Figure 2.10). The objective of an attacker is to control the instruction pointer, hence **code execution**.

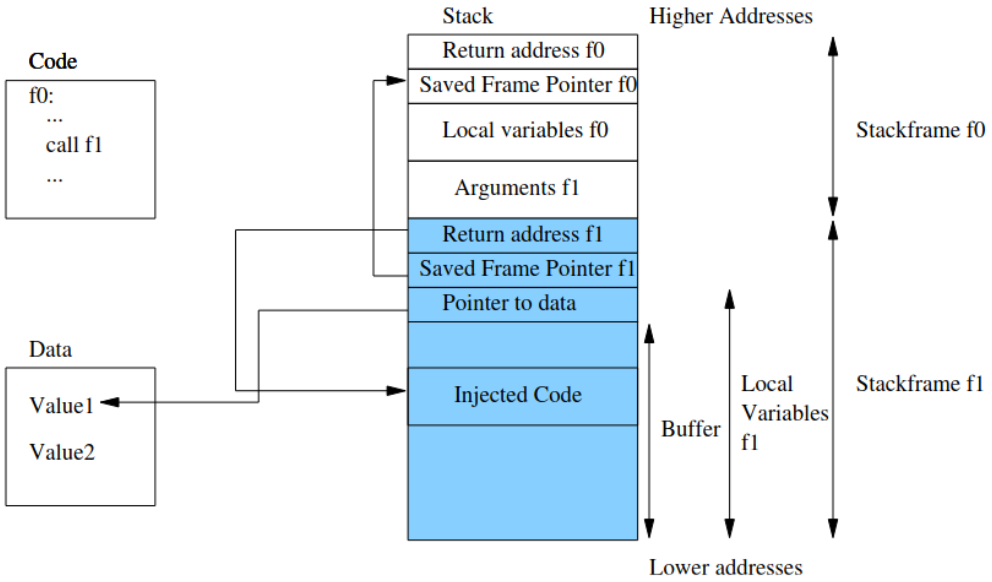


Fig. 2.10. Write return address with a buffer overflow[10]

```c
int main()
{
        char * input = "PASSWORD12";
        char buffer[8];

        strcpy(buffer,input);

        printf("%s",buffer);
        return 0;
}
```

Listing 2.1. Buffer overflow example in C

Note that in this section the simplest scenario of a buffer overflow is explained to understand the vulnerabilities found in this work. However, in modern systems there are some protections and mitigation techniques like the use of stack cookies, DEP, ASLR, etc.(See Section 2.3.2). These protection techniques are not perfect though, with new security measures, new exploitation techniques appear, like Return-Oriented programming.

Also, there is another memory structure apart from the Stack, this is the heap. The idea is the same, but instead of overwriting a return address, the objective is to overwrite function pointers that are located after the buffer. Function pointers are not always available though.

2.5. Prior research

In this section, the most important works related with the research in Shannon baseband are presented.

2.5.1. RAMDUMP and Progress Message

Until Nico Golde and Daniel Komaromy presented their work in the security conference REcon 2016 [11][12], all the research done in baseband processors was done on Qualcomm basebands. In this conference they explain how they found a Buffer Overflow in the Shannon baseband processor.

To perform their research, they started analyzing the *modem.bin* file from the phone firmware. This file contains all the code of the baseband processor. Although the header of this file does not look standard there is a pattern that seems as an array of structs that describes the different segments of the firmware. Figure 2.11.

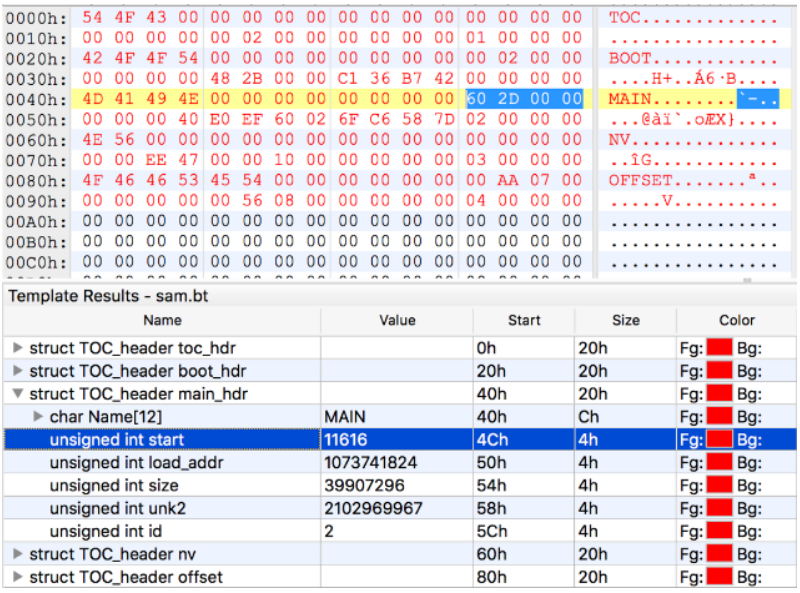


Fig. 2.11. Header of modem.bin file[12]

Analyzing the BOOT segment, they found out that the code was actually valid ARM code. It was not the case for the MAIN segment. The reason is that this segment was encrypted. They confirmed it by measuring the entropy of the MAIN region. Entropy is

a variable to measure the amount of the information or disorder of a set of data. High entropy means that the information is very likely to be compressed or encrypted.

These researchers discovered that it is possible to get the code once loaded in memory in plain text using RAMDUMPs. This means, once the MAIN segment of the code(not-encrypted) is loaded in memory, copy all the data into a file. This file will contain all the information in memory at that specific time, including the MAIN section in plain text.

The non-encrypted MAIN segment obtained from the RAMDUMP was loaded into a disassembler/debugger to be studied. They identified important parts in the firmware to have an overall idea of how it works.They found the task scheduler and important functions to manage memory, like the MPU configuration using the *MCR* instructions. (See Section 2.3.1)

As mentioned in Section 2.2 to implement RADIO protocols, packages are being received and sent constantly. An incoming message will be processed as follows: First, once arrived at the device, this is stored in a queue, then another task will decode a specific type of message, parsing each field of the package in an internal structure, and later another task will process the message, generating the appropriate response to that message.

Dequeue message → Decode message → Trigger response

Nico and Daniel found a vulnerability in the decode task that parses the *progress* message. Figure 2.12. From the standard specification about the progress message:[2]

> *This message is sent from the network to the mobile station to indicate the progress of a call in the event of interworking or in connection with the provision of in-band information/patterns.*

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

Fig. 2.12. PROGRESS message content[2]

This was literally a "text book" buffer overflow(See Section 2.4). The vulnerability relies on the Progress Indicator field having a LV format. What means *Length-Value*. The

function in charge of decoding this message is trusting the Length attribute which leads to a stack overflow (even when this field has always a static length of 3). This vulnerability identifier is *CVE-2015-8546*[13] which has an overall CVSS Score of 9.8 over 10. Note that CVSS is a system for assessing the severity of computer system vulnerabilities, assigning a score from 0 to 10.A score of 9.8 is considered Critical. Figure 2.13 shows the code where this bug is introduced. Note how the length value is used in function *memcpy* without bounds check, causing the buffer overflow.

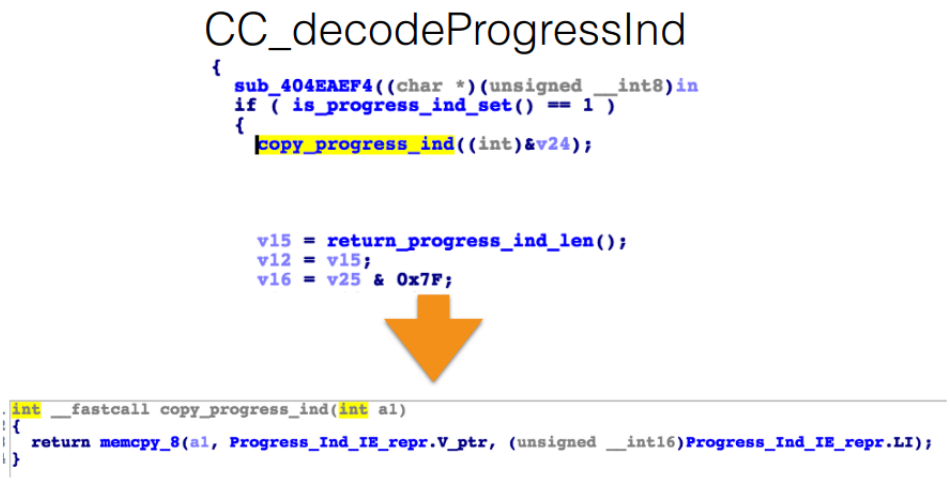


Fig. 2.13. CVE-2015-8546 Progress Indicator Bug[12]

With this buffer overflow, and the few mitigations inside this baseband processor, an attacker can overwrite a return address, leading to code execution in the baseband processor and since the whole code is running in kernel mode the exploit will have the maximum privileges.

2.5.2. PDP context accept vulnerability

In Tokyo, was the Pwn2Own, a computer hacking contest held at the CanSecWest security conference where the participants are challenged to exploit software with previously unknown vulnerabilities. During this contest, Amat Cama discovered another bug in Shannon's baseband. In his presentation [14] he explained the vulnerability and the process we followed in order to find it.

He decided that instead of looking for bugs in GSM, he will focus on GPRS since this protocol is much more complex than GSM and is a less explored area of research, therefore, the chances of finding a bug are higher.

As we know, the messages of this protocol are composed of Information Elements (IE).In addition to the Value field, these information elements can contain a header indicating the

Type and the Length of the following value. The presence of these headers is described in the official documentation of the protocol [2].

The IE more relevant from the point of view of an attacker are the TLV(Type, Length, Value) and the LV(Length, Value). This is because in the IE that contains a length header, there are chances that the code that parse these packets, relies on this length value, what could lead to memory corruption bugs, just like it happened in the bug found by Nico and Daniel, described in Section 2.5.1.

Reading the standards, he found the ACTIVATE PDP CONTEXT ACCEPT message. According to the documentation, this message is sent by the network to the MS to acknowledge activation of a PDP context. PDP stands for Packet Data Protocol, in short, is a message to manage the IP (Internet Protocol). In particular, the Protocol Configuration options parse the IP control protocol headers.

Inside this task there is one part that seems interesting, this is the parsing of the *Protocol Configuration options*. This information element is a TLV with a length of 3-253 bytes, as we can see in Figure 2.14.

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

Fig. 2.14. ACTIVATE PDP CONTEXT ACCEPT message content [2]

Inside the function processing this Information Element there is a loop that copies data to the stack relying on some fields of our data. Figure 2.15 shows the vulnerable code. The buffer is overflowed since the loop is relying on the *len* variable containing the length value of the TLV.

```
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;
    }

    ...
}
```

Fig. 2.15. Buffer overflow in the copy of Protocol Configuration options [14]

The problem of this bug is that if a crafted message is sent to the phone directly, this will ignore it, due to, as said in Section 2.3.2, the phone has a set of state machines. Therefore, this needs to be in the correct state before processing the message, in order to be in the correct state, the phone needs first to send the ACTIVATE PDP CONTEXT REQUEST to the network. Reading the documentation Amat discovered that it is possible to force the phone to send this message if the network first sends the REQUEST PDP CONTEXT ACTIVATION message. This way, the state machine will move to the state needed in order to decode our crafted message and trigger the bug. Figure 2.16

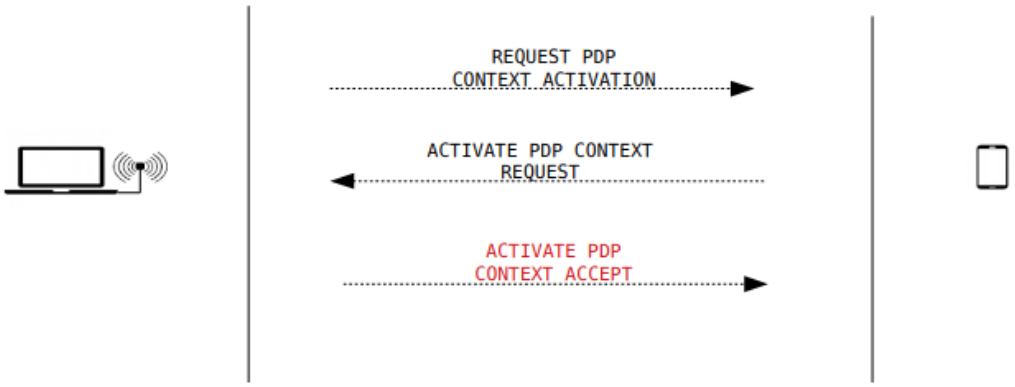


Fig. 2.16. Messages sent in order to trigger Amat's bug [14]

Note that although this bug can look very straightforward we need to take into account the baseband operating system is a very complex system, with over 70 thousand functions, the process of locating this type of vulnerabilities is not an easy task.

2.5.3. Firmware emulation

Grant Hernandez & Marius Muench came up with another idea to find vulnerabilities in Shannon, developing a tool called ShannonEE (not publicly released yet) [15] [1], in

short, this platform can emulate Samsung's baseband in order to execute the firmware directly without the need of a physical phone.

The motivation behind this project relies on the fact that Shannon is a very complex Operating System, therefore, performing Static Analysis require a lot of effort and time as well as some experience with reversing engineering, and perform OTA live fuzzing is not a good solution since the basebands are very fragile and after some corruptions they do not behave as expected.

Thus, an emulated firmware modified to include a task that acts as fuzzer that sends random data automatically to the target tasks as if the data is being sent over the air looks like a very attractive solution. It does scale very well and all the process can be automated, also, having the software running in a platform the researcher has full control of, provides the ability of introducing debugging capabilities and modifying parts of the firmware before running it, in contrast with the restrictions in a physical phone, like firmware signature and limited RAMDUMPS or logs(depending on if the phone is rooted or not).

However, emulating an embedded system like Shannon is not an easy task. The main problem is that this operating system is dependent on the hardware and the software relies on peripherals in order to work properly, this means that this software will expect a specific behavior from peripherals, for example a response to a read request. If the firmware does not get any response it will crash and not run at all. In particular with Shannon, the function *pal_init1()* is called to initialize all modem subsystems and tasks, including most of the peripherals. Once they managed to partially emulate all these peripherals the emulator was ready. These peripherals were just partially emulated since they do not need to make them work exactly as the hardware components do. They just need to provide a valid response to the firmware in order to pass the initial checks and make the firmware work properly.

With the firmware emulated they used Triforce-AFL[16] to include what they called an AFL task that will target the task messages queues, this is a much simpler and better approach than targeting the radio controller, just the raw message needs to be specified and be sent to the queue, this will be decoded and processed. The process will be repeated iteritavility, expecting that some messages will crash the firmware. Figure 2.17

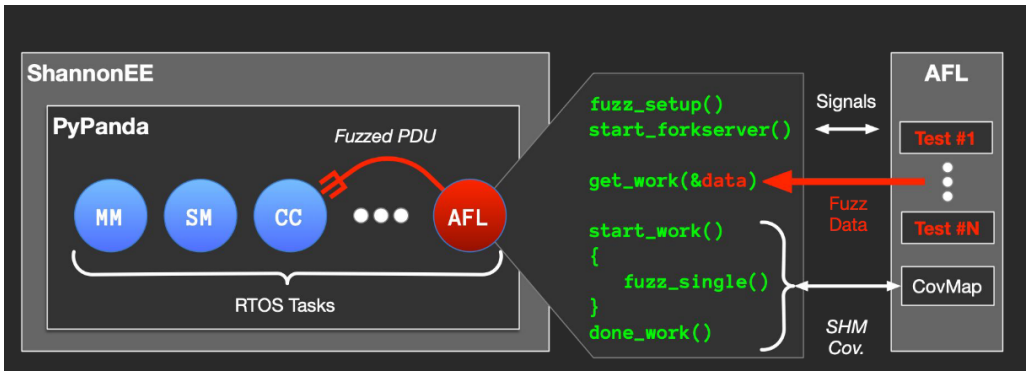


Fig. 2.17. AFL Fuzzes the message queues of CC tasks. [1]

After running this tool for 5 days in 30 CPU cores, it automatically rediscovered the *GPRS PDP_NETWORK_ACCEPT* bug reported by Amat (See Section 2.5.2) and even a 0-day was found and reported to Samsung. This vulnerability, *SVE-2020-18098* is one of the vulnerabilities that is hunted in this work (See Section 3). where different techniques are used in order to find and exploit the bug with the information provided in the vulnerability disclosure.

3. APPROACH

In this chapter, the approach carried out by the author to find a vulnerability in a Samsung baseband is explained in detail. Note that the concepts described in this section can be used for any researcher that aims to reverse engineer any kind of software having little knowledge about the technology used. The main objective when reversing a software is not to understand everything, but to understand the most important things to find the vulnerability.

3.1. Static Analysis

In this section the investigation is carried out using Static Analysis, this means that the code is analyzed without running it.

3.1.1. Load firmware

We can refer to Sammobile [9] to download almost any Samsung firmware. The firmware of the device is a *.zip* that contains different *tar.md5* files. The file that starts with *CP* followed by the code of the device, is the firmware that is installed in the *Communication Processor* or *Baseband*. Inside there is the *modem.bin*, the binary file that is running on the baseband processor ARM-Cortex-R7.

With the help of an hex editor we can see in the Figure 3.1 the header of the file *modem.bin*. There, we can identify some Strings that correspond to different sections inside this firmware. In this case we are interested in the *MAIN* part of the modem. Following the magic word corresponding to the section, it follows an integer (represented in little endian) that corresponds to the offset of that specific part. For the MAIN section, the value of this offset. *c0 1c 00 00* → *00 00 1c c0* (Little Endian).

```
00000000: 544f 4300 0000 0000 0000 0000 0000 0000  TOC.............
00000010: 0000 0000 0002 0000 0000 0000 0500 0000  ................
00000020: 424f 4f54 0000 0000 0000 0000 0002 0000  BOOT............
00000030: 0000 0040 a81a 0000 989d 0d0b 0100 0000  ...@............
00000040: 4d41 494e 0000 0000 0000 0000 c01c 0000  MAIN............
00000050: 0000 0140 6850 1902 6e10 4583 0200 0000  ...@hP..n.E.....
00000060: 5653 5300 0000 0000 0000 0000 406d 1902  VSS.........@m..
00000070: 0000 9046 ac69 2300 0b6b b74c 0300 0000  ...F.i#..k.L....
00000080: 4e56 0000 0000 0000 0000 0000 0000 0000  NV..............
00000090: 0000 3046 0000 1000 0000 0000 0400 0000  ..0F............
000000a0: 4f46 4653 4554 0000 0000 0000 00aa 0700  OFFSET..........
000000b0: 0000 0000 0056 0800 0000 0000 0500 0000  .....V..........
```

Fig. 3.1. Modem.bin header. Highlighted the address, the loading address and size of the MAIN region

The next step is to create a new file *modem_main.bin* containing just this main section in order to make the analysis easier. We proceed using the tool **dd** specifying the address of the main section obtained earlier.

```
dd if=modem.bin of=modem_main.bin bs=$((16#1cc0)) skip=1
```

Now we can add modem_main.bin to our disassembler, (IDA pro). We select the loading address (0x40010000) as specified in the modem header. See Figure 3.1

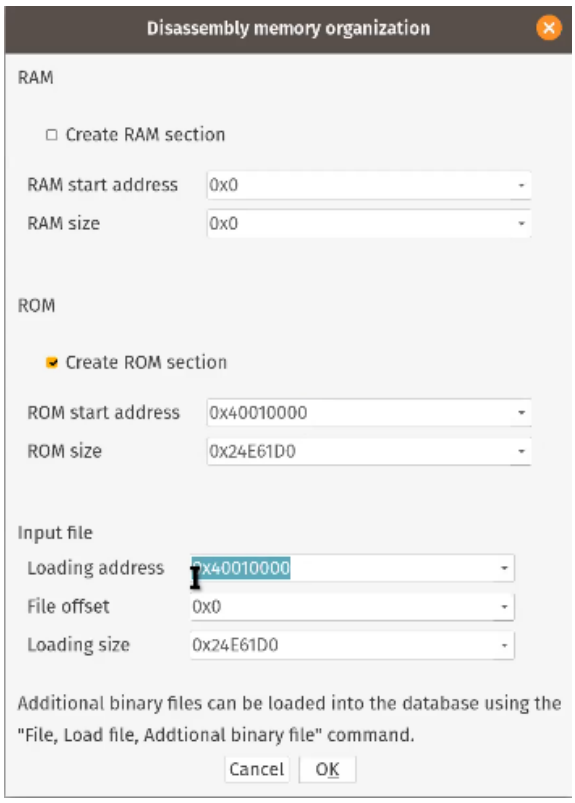
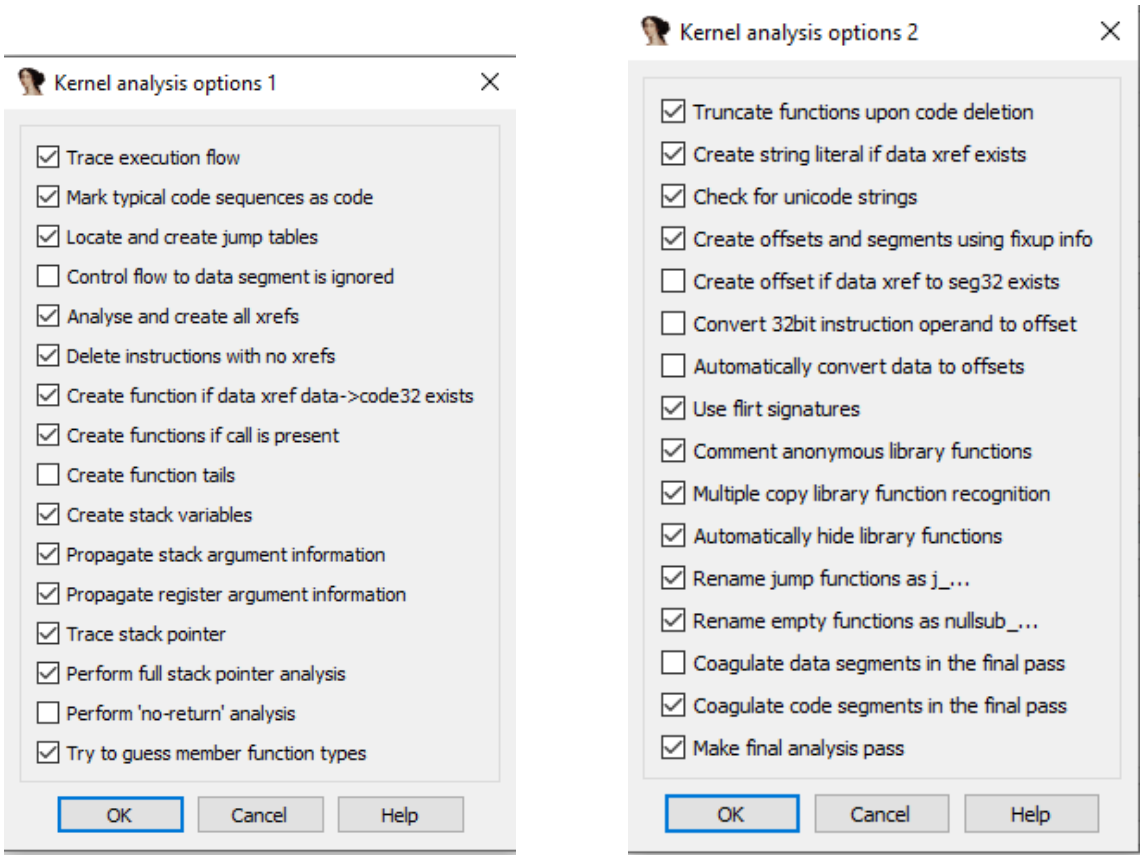


Fig. 3.2. Base address selection

At this point we have our firmware loaded in IDA pro, but in order to analyze this firmware it is convenient to run the IDA Analyzer so the disassembler detects and labels the functions. Before running the analysis it is recommended to change some options in: *Options →General →Kernel options 1 and Kernel options 2.*
For the Kernel options 1 is recommended to turn off *function tails* and *no-return analysis*, in Kernel analysis options 2, it is recommended to turn on *Make final analysis pass* as shown in the Figure [3.3]



(a) Kernel analysis options 1

(b) Kernel analysis options 2

Fig. 3.3. Recommended options to perform the analysis

At this moment we have the firmware loaded into IDA and most of the functions identified, but we can make use of some Python scripts to improve this functions detection and in order to label the functions.

The scripts [17] used are provided by Comsecuris, and specifically made to help the researcher to reverse engineer Shannonn's baseband.

The script *def_arm32_functions.py*[17] finds functions, prologues and epilogues in the firmware to detect functions. A function prologue is a set of specific instructions that are always executed at the beginning of a function to prepare the stack and the registers to use inside the function. In contrast, a function epilogue is a set of instructions that appears at the end of the function to restore the stack and registers. Knowing this, the script looks for these specific patterns to detect functions. Figure 2.11 shows a function header. The first instruction *PUSH.W* is storing in the stack the value of the saved registers *r4-r11* and the register *LR*. This is done so when the function returns the original registers values are restored. See Figure 3.4

```
000 2D E9 F0 4F          PUSH.W      {R4-R11,LR}
024 00 24                MOVS        R4, #0
024 AD B0                SUB         SP, SP, #0xB4
0D8 DF F8 40 A2          LDR.W       R10, =CcCurrentStackId
0D8 FF 26                MOVS        R6, #0xFF
```

(a) ARM Prolog function in Shannon

```
0D8 2D B0                ADD         SP, SP, #0xB4
024 BD E8 F0 8F          POP.W       {R4-R11,PC}
```

(b) ARM Epilogue function in Shannon

Fig. 3.4. Function prologue and epilogue

In addition, it is very useful to label functions with descriptive names, by default, all functions are labelled with the word "sub" followed by the address of the function, e.g *sub_40A4A3B0()*, this gives us no information at first sight about what this function is doing. To solve this problem the python script *label_functions.py*[17] uses the Strings in the firmware along with some heuristics to label the functions.

3.1.2. STATUS message vulnerability

In this part I explain the process I followed to find a n-day vulnerability in this firmware, this vulnerability was previously reported by Amat Cama [18] The information we have is that the vulnerability is a buffer overflow as the vulnerabilities explained in Section 2.5 and that is due to an abnormal STATUS message.

Knowing this and having our environment setted up, the next step is to locate the function that is in charge of parsing the message, as explained in Section 2.5, this function decodes the raw message coming from the air and places it into an internal structure in the memory.

To find this function we can look at the structure that stores the linked list with all the tasks or we can identify this function from the Strings available in the firmware and back trace where this String is used to finally identify the function.

In the Strings window of Ida all the Strings along with the memory address that contains it. Since we want to locate the function in charge of decoding the STATUS message we search the word *status* in the Strings tab. Among all the results there is one String that looks relevant.Figure 3.5

```
ROM:40492E30  0000001D  C  MM <== <RADIO MSG> MM_STATUS
ROM:40493250  0000001D  C  MM ==> GRR_MM_LTE_STATUS_IND
ROM:404939A8  0000001A  C  MM ==> <RADIO MSG> STATUS
ROM:40494118  00000020  C  MM ==> URRC_MM_EUTRA_STATUS_IND
ROM:404944C4  00000025  C  MM ==> MM_NS_REGISTRATION_STATUS_IND
ROM:404946EC  00000021  C  MM ==> MM_NS_HANDOVER_STATUS_IND
ROM:40495420  00000035  C  MM ==> MM_NS_REGISTRATION_STATUS_IND[CS = 0, PS = 0]
ROM:40495458  00000035  C  MM ==> MM_NS_REGISTRATION_STATUS_IND[CS = 1, PS = 0]
ROM:40495490  00000035  C  MM ==> MM_NS_REGISTRATION_STATUS_IND[CS = 0, PS = 1]
ROM:404954C8  00000035  C  MM ==> MM_NS_REGISTRATION_STATUS_IND[CS = 1, PS = 1]
ROM:40495F58  00000043  C  Unknown Rat Change Window State in mm_GetRatChangeWindowStatusName
ROM:404961B0  00000017  C  GMM_NO_IMSI_UPD_STATUS
ROM:40496F44  00000026  C  NASL3.PLMN.Samsung SOR Feature Status
ROM:40497CB4  0000001C  C  CC <== CC_VTCALL_STATUS_REQ
ROM:40498090  00000021  C  CC <== <RADIO MSG> STATUSENQ_IND
ROM:404980D4  0000001E  C  CC <== <RADIO MSG> STATUS_IND
ROM:40498854  0000001E  C  CC ==> <RADIO MSG> STATUS_REQ
ROM:40498874  00000022  C  CC ==> <RADIO MSG> STATUS_ENQ_REQ
ROM:4049A2BC  00000026  C  SM <== GMMSM_PLMN_COVERAGE_STATUS_IND
ROM:4049A2E4  0000001C  C  SM <== GMMSM_PDP_STATUS_IND
ROM:4049A430  00000019  C  SM <== RABMSM_STATUS_REQ
ROM:4049A49C  00000017  C  SM <== SNSM_STATUS_REQ
ROM:4049A6B4  0000001D  C  SM <== <RADIO MSG> SM_STATUS
ROM:4049A8B0  00000020  C  SM ==> GMMSM_PDP_STATUS_UPD_REQ
ROM:4049AB90  0000001D  C  SM ==> <RADIO MSG> SM_STATUS
ROM:4049D0F4  00000018  C  SS_GAPI_SS_ERROR_STATUS
```

Fig. 3.5. Strings window IDA Pro

The arrow pointing to the left could mean that this Strings reference a message coming from the network to the Mobile Station. CC means Call Control, therefore, that string is what we are looking for. Clicking on the entry leads us to the address where this String is located. Figure 3.6

```
ROM:404980B3 00                          DCB    0
ROM:404980B4 43 43 20 3C+    aCcRadioMsgProgress_ind DCB "CC <== <RADIO MSG> PROGRESS_IND",0
ROM:404980B4 3D 3D 20 3C+                                    ; DATA XREF: ROM:401F881C↑o
ROM:404980D4 43 43 20 3C+    aCcRadioMsgStatus_ind DCB "CC <== <RADIO MSG> STATUS_IND",0
ROM:404980D4 3D 3D 20 3C+                                    ; DATA XREF: ROM:401F894C↑o
ROM:404980F2 00                          DCB    0
ROM:404980F3 00                          DCB    0
ROM:404980F4 43 43 20 3C+    aCcRadioMsgConnect_ind DCB "CC <== <RADIO MSG> CONNECT_IND",0
ROM:40498113 00                          DCB    0
```

Fig. 3.6. Debug String located in memory

In this address we can see that there are many strings all together. Each of them looks to have one struct, containing the String, followed by a few 0 Bytes.In this struct, we see that there is a DATA XREF, this is a feature in IDA, thata gives us the reference of where that part of memory is referenced from. Since, what we want is to find the code that processes that message, we follow that XREF.

In this structure we can see that there is a pointer to a function, followed by some values *0x10000B and 0x2A3C*, and then a pointer to the table containing the String. The python scripts used already labeled the function as *cc_DecodeStatusIndMsg*. By the name, this means that this function is decoding the STATUS message, exactly where the bug could have been introduced.

Inside the function we can see the pseudocode of the function that is over 500 lines. At first sight the code looks complicated, there are many functions, the variables do not have descriptive names, they have the format of v1, v2, v3...

Until now, we know that it is very likely that there is a bug in the code and this function is in charge of decoding the message and storing the fields in memory. In order to learn what fields must be decoded it is necessary to refer to the GSM documentation [2].

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

Fig. 3.7. STATUS message content. [2]

Figure 3.7 shows the fields in the STATUS message. The most relevant are *Cause* and

Auxiliary states since these are LV (Length Value) and TLV (Type Length Value). In the first part of the pseudocode some checks are done to the fields. First it is checked if the Cause is available, then it is checked if the value the Cause is greater than zero. If it is zero a log is printed: *cc_DecodeStatusIndMsg - invalid CcCauseLen = 0 received*.

However, the function seems to get this value directly from memory, and not computing the real value of the cause. This value retrieved from memory matches with the Length header. But, in order to have memory corruption it is needed to find a place in the code that relies on this value and overwrite memory.

The function called *cc_copy_cause_ie* called my attention. If this function is relying on the Length header of the cause to perform the copy means that there is stack buffer overflow. In effect, this is the case. The function mentioned above calls a copy function passing as parameter the length of the header from the LV. (See Figure 3.8)

```
int __fastcall cc_copy_cause_ie(void *a1)
{
  return cc_CopyBufferBytes(
           a1,
           *((void **)&cc_IE_base.cc_cause_ie + 94 * (unsigned __int8)CcCurrentStackId),
           *((unsigned __int16 *)&cc_IE_base.cc_cause_ie_len + 188 * (unsigned __int8)CcCurrentStackId));
}
```

Fig. 3.8. Vulnerable function causing the Buffer Overflow

3.1.3. SETUP message vulnerability

In this section we locate the vulnerability found by Grant Herndandez and Marius Muench emulating Samsung's firmware. This vulnerability was assessed as critical and it consists of Buffer Overflow in the baseband that allows arbitrary code execution. Figure 3.9

SVE-2020-18098: Buffer overflow vulnerability in Baseband with abnormal SETUP message

Severity: Critical
Affected versions: Devices with select Exynos modem chipsets
Reported on: June 19, 2020
Disclosure status: Privately disclosed.
A possible buffer overflow vulnerability in baseband allows arbitrary code execution.
The patch adds the proper validation of the buffer length.

Fig. 3.9. SVE-2020-18098. SETUP buffer overflow vulnerability

As done with the previous vulnerability, the first step is to locate the function that decodes the SETUP message, for this, we proceed as in the previous vulnerability. First we locate the the string *CC <== <RADIO MSG> SETUP_IND* in the Strings window in IDA, from there we go to the function *cc_DecodeSetupIndMsg*. Once decompiled in IDA, this function is composed of 989 lines of pseudocode. In the documentation (See Figure 3.10) we can see all the Information Elements that are sent in this message. Most of them are TLV and optional (labelled with "O" in the column "Presence").

Table 9.70/3GPP TS 24.008: SETUP message content (network to mobile station direction)

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
	Setup Message type	Message type 10.4	M	V	1
D-	BC repeat indicator	Repeat indicator 10.5.4.22	C	TV	1
04	Bearer capability 1	Bearer capability 10.5.4.5	O	TLV	3-16
04	Bearer capability 2	Bearer capability 10.5.4.5	O	TLV	3-16
1C	Facility	Facility 10.5.4.15	O	TLV	2-?
1E	Progress indicator	Progress indicator 10.5.4.21	O	TLV	4
34	Signal	Signal 10.5.4.23	O	TV	2
5C	Calling party BCD Number	Calling party BCD num. 10.5.4.9	O	TLV	3-14
5D	Calling party sub- Address	Calling party subaddr. 10.5.4.10	O	TLV	2-23
5E	Called party BCD Number	Called party BCD num. 10.5.4.7	O	TLV	3-19
6D	Called party sub- Address	Called party subaddr. 10.5.4.8	O	TLV	2-23
74	Redirecting party BCD number	Redirecting party BCD num. 10.5.4.21b	O	TLV	3-19
75	Redirecting party sub-address	Redirecting party subaddress. 10.5.4.21c	O	TLV	2-23
D-	LLC repeat indicator	Repeat indicator 10.5.4.22	O	TV	1
7C	Low layer Compatibility I	Low layer comp. 10.5.4.18	O	TLV	2-18
7C	Low layer Compatibility II	Low layer comp. 10.5.4.18	C	TLV	2-18
D-	HLC repeat indicator	Repeat indicator 10.5.4.22	O	TV	1
7D	High layer Compatibility i	High layer comp. 10.5.4.16	O	TLV	2-5
7D	High layer Compatibility ii	High layer comp. 10.5.4.16	C	TLV	2-5
7E	User-user	User-user 10.5.4.25	O	TLV	3-35
8-	Priority	Priority Level 10.5.1.11	O	TV	1
19	Alert	Alerting Pattern 10.5.4.26	O	TLV	3
2F	Network Call Control Capabilities	Network Call Control cap. 10.5.4.29	O	TLV	3
3A	Cause of No CLI	Cause of No CLI 10.5.4.30	O	TLV	3
41	Backup bearer capability	Backup bearer capability 10.5.4.4a	O	TLV	3-15

Fig. 3.10. SETUP Message content

In the code I identified some patterns, and parts of the code that matches the decode of these Information elements. However, many other auxiliary functions are called, growing exponentially the complexity of the code. IDA and the scripts used to identify functions help but are not perfect, most of the functions called are not labelled and are composed by hundreds of lines of pseudocode, that would need to be studied in order to understand the original calling function. Because of that I used the Binary diffing technique, as it is presented below.

Binary diffing

Binary diffing is a Reversing Engineering technique where two binary files are compared

to detect the differences between them. In this part of the research two different versions of the firmware were downloaded. One version with the security patch level prior to the vulnerability disclosure(October 2020) and another one with the security patch level posterior to the vulnerability patch.

As both binary files belong to the firmware of the same phone model and very close in time, the difference between both should be the patches or features introduced in the new version. Since these patches and features are significantly less code than the entire firmware, the goal is to find the part of the code that patches the vulnerability we are looking for. Since this is a memory corruption bug. The fix could be a simple *if-else statement* checking the length of a buffer, or maybe some assumptions regarding the type of the data received over the network.

The first approach carried out to perform this binary diffing is to run a tool named Diaphora, this program works as an IDA Plug In. It first creates an .sqlite database for each of the firmwares, then using some heuristics it detects similarities between functions. After 15 hours looking for similarities. Diaphora found a total of 4829 exact matches, 2836 unreliable matches and 4739 Partial matches.

There are much more differences than expected between both versions, even when these correspond to the same phone and are different by just a few months. Some techniques could be used at compilation time for efficiency reasons that change the structure of the firmware that much. Or it could be the case that IDA does not detect all the functions in one firmware and it does in the other.

I spent some time manually analyzing some functions in the Partial matches that could be of interest, like copy-like functions and others related to the SETUP message looking for a noticeable difference. However, the patch could be a simple conditional statement that does not allow a variable to have a value greater than a specific one. The process of manually finding the lines of code that perform this check in this type firmware having the results we have from the binary diffing is not doable since the lines performing this check could be one among hundred thousands of lines of pseudocode. Even when one difference in a conditional statement is found between both versions, the process of understanding the meaning of that variable, where is modified and where is used in order to know if is of our interest or if is it even related to the bug we are looking for, gets extremely difficult to analyze due to we do not have the source code and the system itself is very complex.

Instead of diffing the entire firmware I took the approach of diffing just the setup function. As expected, diaphora identified this function as a partial match. In the assembly and pseudocode the names of registers and variables are different to each other, which makes it more difficult to diffing line by line. I found out that it is better to analyze the differences in the graph view, checking if there is a significant change in the execution flow due

to the checking of the buffer length, but nothing relevant was found. As explained, most likely, the bug is introduced in a function called by the *cc_DecodeSetupIndMsg()*, but not directly in that function.

Figure 3.12 and Figure 3.11 show the graph view of the *cc_DecodeSetupIndMsg()* function in the unpatched and patched version respectively.

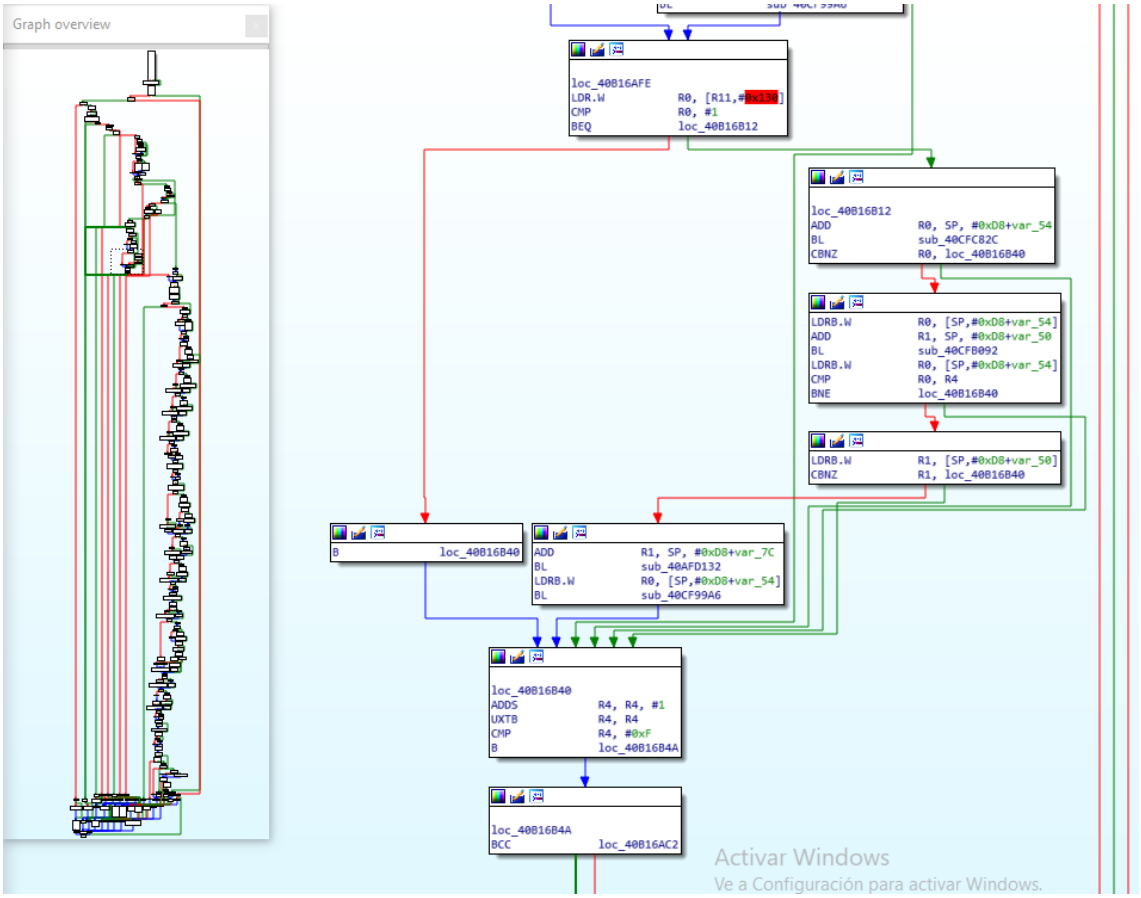


Fig. 3.11. Graph view of the decode setup message function. Patched version

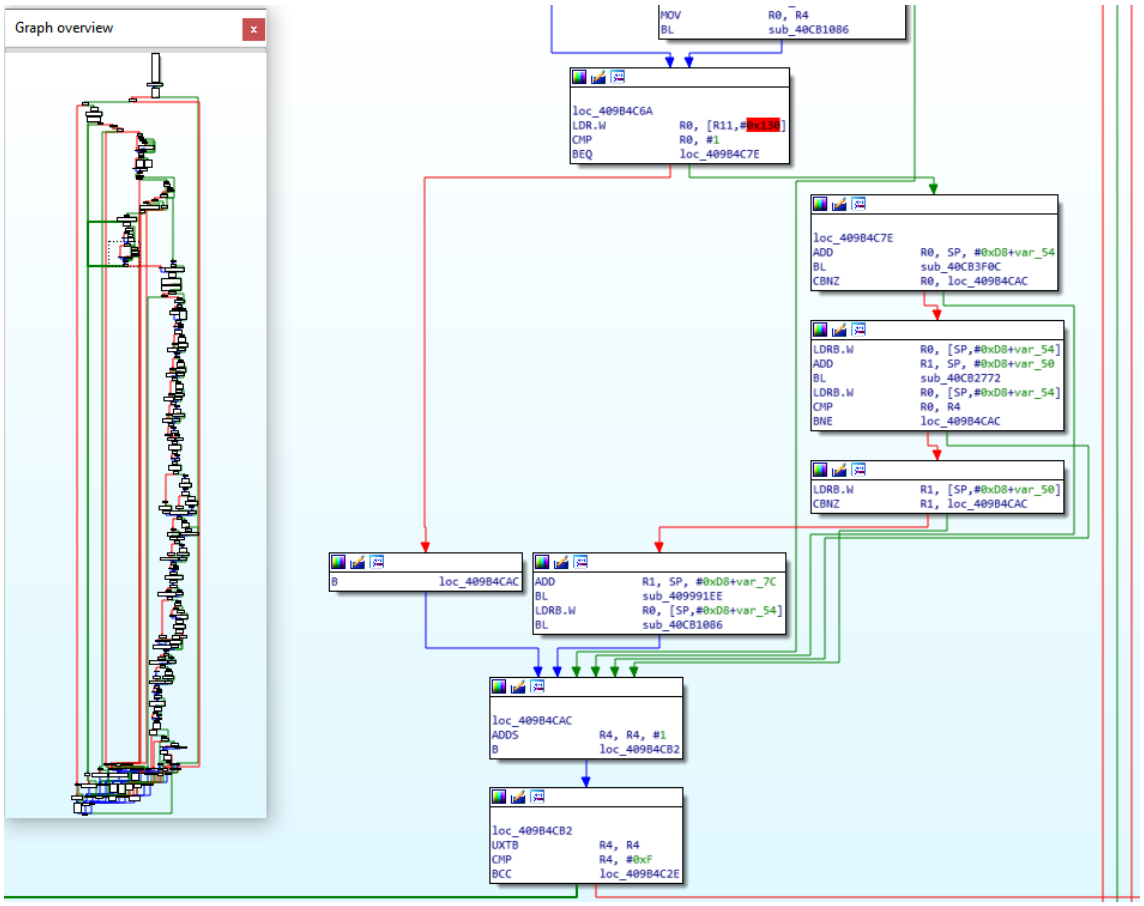


Fig. 3.12. Graph view of the decode setup message function. Unpatched version

In depth firmware analysis

Diffing approach did not succeed, therefore more work is needed in order to understand the structure of the setup message. After some tries, I found out that the most efficient way to understand the structure of the function is to compare it with the functions that decode other messages. This way I realized that some patterns repeat for both functions. First, a lot of variables are created and some lines of code are similar for both setup decodes. These should be some initial checks before parsing the fields of the incoming package. First, we can see how a new variable is created storing the pointer of the setup package without the header that specifies the package as a CC message. Then it is checked that the eight bit of the first byte has a zero value using a mask with *0x80*. Otherwise the functions returns.(See Figure 3.13)

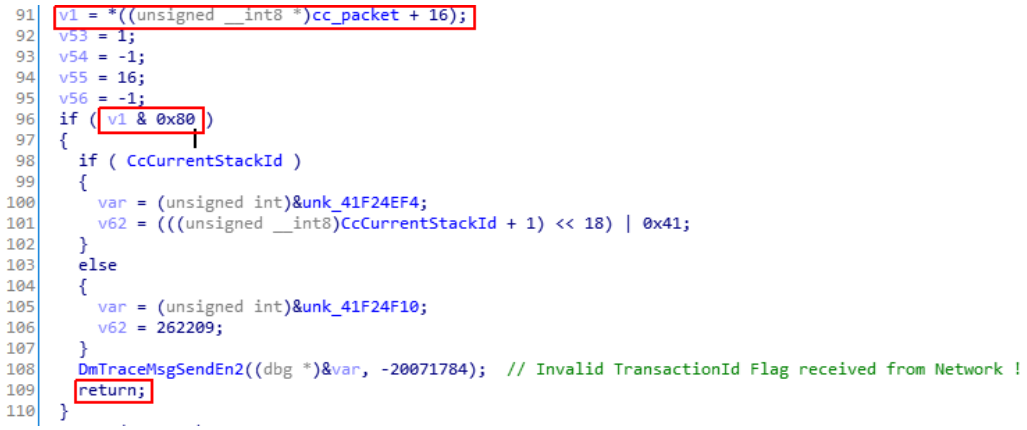


Fig. 3.13. Check that the 8th bit of the first Byte has a zero value

Section 10.4 of the GSM documentation[2] explains the reason for this check.

> *When the radio connection started with a core network node of earlier than R99, bit 8 shall be set to 0 and bit 7 is reserved for the send sequence number in messages sent from the mobile station. In messages sent from the network, bits 7 and 8 are coded with a "0". See 3GPP TS 24.007.*
>
> *When the radio connection started with a core network node of R'99 or later, bits 7 and 8 are reserved for the send sequence number in messages sent from the mobile station. In messages sent from the network, bits 7 and 8 are coded with a "0". See 3GPP TS 24.007.*

The next step is to identify within the code most of the information elements as possible. To do this, the decompiled code is copied to a text editor (Visual Studio Code) in order to perform the needed annotations, and variables renaming without modifying the original code.

Before applying the needed operations to each of the information elements, first, a function is called to check if the Information Element is included since most of the Information Elements are optional for this package (See Figure 3.14).

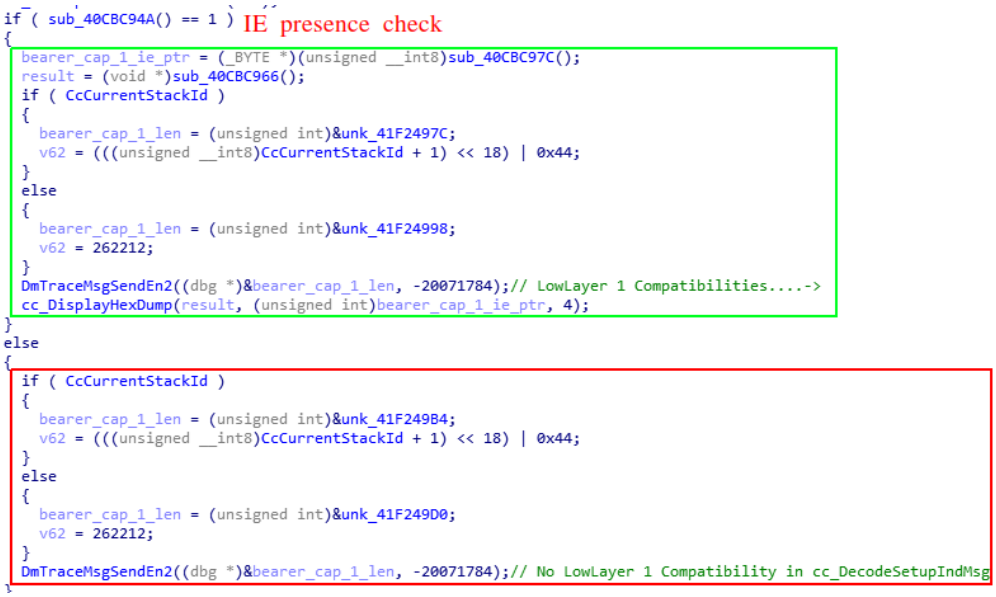


Fig. 3.14. Example of code checking the presence of an information element

This function checks if the element is included in the message checking the data in the stack and calculating the offset of the data that is being checked along with the GSM specification, it is possible to know where a specific element is being checked.

Once we have most of the Information elements identified in the code. It is needed to study each Information Element in the code along with the specification in order to find out what parts of these fields contain variable elements specified in the message that can lead to a potential memory corruption using a crafted package.

According to the specification the *BC repeat indicator* field has the letter C in the presence column (See Figure 3.10) and as described also in the documentation this field needs to be included **if and only if** bearer capability 1 and bearer capability 2 IE are included: [2]

> *The BC repeat indicator information element is included **if and only if** bearer capability 1 information element and bearer capability 2 IE are both present in the message.*

For the Bearer capability 1 and 2, there are also some constraints defined in the documentation.

> *The bearer capability 1 information element may be omitted in the case where the mobile subscriber is allocated only one directory number for all services (ref.: 3GPP TS 29.007 [38]). The bearer capability 2 IE is missing **at least if** the bearer capability 1 IE is missing.*
>
> *If the MSC wishes to indicate capability for an alternative call mode, which can be entered through fallback, this is indicated by adding a bearer capability information element (bearer capability) 2 element (see subclause 5.3.6).*

Those constraints called my attention. *if and only if* means that if one of the fields is included it is expected that the other one is included too. My hypothesis is that the Engineers who wrote that code could rely on this implication and take for granted that the other field will exist without checking it. Therefore, a crafted message that does not follow these constraints, for example, including the *BC repeat indicator* and the *bearer capability 1* (Without bearer capability 2) will trigger the bug.

The underlying idea is that the program checks first if the *BC repeat indicator* field is included and assumes that if this field exists, the *bearer capability 2* will be included as well. It will perform some operations in the memory buffer where *bearer capability 2* is expected to be. And since this field was not added in our crafted package, that memory buffer could be occupied by other data that would be overwritten. For example, the length value of another IE, causing the buffer overflow later.

After studying the code that is involved with this bearer capabilities in detail. I found that there is a function that checks if *BC repeat indicator* is available (*sub_40CBC308*). Figure 3.15a

```
if ( sub_40CBC308() == 1 )
  v29 = cc_get_repeat_ind();
else
  v29 = 0;
v49 = v29;
cc_GetRepeatIndicatorName(v29);
```

(a) Get BC Repeat Indicator

```
void __fastcall cc_GetRepeatIndicatorName(unsigned int a1)
{
  const char *repeat_indicator; // r1
  void *v2; // [sp+0h] [bp-10h]
  int v3; // [sp+4h] [bp-Ch]

  if ( CcCurrentStackId )
  {
    v2 = &unk_41F14578;
    v3 = (((unsigned __int8)CcCurrentStackId + 1) << 18) | 0x44;
    if ( a1 < 5 )
      goto LABEL_6;
LABEL_7:
    repeat_indicator = "Unknown RepeatIndicator in cc_GetRepeatIndicatorName";
    goto LABEL_4;
  }
  v2 = &unk_41F14594;
  v3 = 262212;
  if ( a1 >= 5 )
    goto LABEL_7;
LABEL_6:
  repeat_indicator = (const char *)off_415C8C14[a1];
LABEL_4:
  DmTraceMsgSendEn2((dbg *)&v2, repeat_indicator);
}
```

(b) Function cc_GetRepeatIndicatorName()

```
; _DWORD off_415C8C14[5]
off_415C8C14    DCD aCc_invalid_repeat_indicator
                                        ; DATA XREF: sub_40A3B3B2+4↑o
                                        ; cc_GetRepeatIndicatorName+A↑o ...
                                        ; "CC_INVALID_REPEAT_INDICATOR"
                DCD aCc_circular_for_successive_selection ; "CC_CIRCULAR_FOR_SUCCESSIVE_SELECTION"
                DCD aCc_repeat_indicator_reserved ; "CC_REPEAT_INDICATOR_RESERVED"
                DCD aCc_service_change_and_fallback ; "CC_SERVICE_CHANGE_AND_FALLBACK"
                DCD 0
; seg003        ends
```

(c) Vector off_415C8c14 containing the BC Repeat Indicator names

Fig. 3.15. BC Repeat Indicator decoding

The first function *cc_get_repeat_ind()* gets an integer value from the stack. And the

function *cc_GetRepeatIndicatorName()*(See Figure 3.15b) uses this value to print in the logs the BC repeat indicator name. The different logs that can be printed depending on the value are the following:(See Figure 3.15c)

- "CC_INVALID_REPEAT_INDICATOR"
- "AR_FOR_SUCCESSIVE_SELECTION"
- "CC_REPEAT_INDICATOR_RESERVED"
- "CC_SERVICE_CHANGE_AND_FALLBACK"

What corresponds to the section 10.5.4.22 of the GSM documentation. [2]. As we can see, the value is printed and not used in any other part of the function that can trigger the bug. Therefore, this field is not likely to be related to the vulnerability.

After some time analyzing the code, running scripts to simplify the code as well as using more binary diffing techniques with different Samsung firmware, I decided to take one step back and read the vulnerability disclosure to start from the beginning with all the knowledge about how the firmware and the GSM protocol works.

The vulnerability description states that it has a buffer overflow via an abnormal SETUP message. The buffer overflow could be stack or heap based. Until this point I assumed that it was a stack based buffer overflow, but, since I did not find anything relevant, I started to look for a heap based buffer overflow. That is why I decided to dig deeper into how memory is managed in ShannonOS. As explained in Section 2.3.2, Shannon has the PAL region, that provides an interface to memory management functions. In that region I found the implementation of the *malloc()* functions, that is used to allocate memory in the heap.

```
                pal_MemAlloc_internal                   ; CODE XREF: pal_TmCreateMsgTimer+12↑p
                                                        ; sub_400249C+14↑p ...
000 DF F8 50 C0                 LDR.W           R12, =byte_415CCDE4
000 10 B5                       PUSH            {R4,LR}
008 DC F8 04 C0                 LDR.W           R12, [R12,#(dword_415CCDE8 - 0x415CCDE4)]
008 BC F1 00 0F                 CMP.W           R12, #0
008 02 D0                       BEQ             loc_400F678
008 BD E8 10 40                 POP.W           {R4,LR}
000 60 47                       BX              R12
                ; ---------------------------------------------------------------------------

                loc_400F678                             ; CODE XREF: pal_MemAlloc_internal+E↑j
008 00 22                       MOVS            R2, #0
008 43 21                       MOVS            R1, #0x43 ; 'C'
008 25 20                       MOVS            R0, #0x25 ; '%'
008 FA F7 14 EA                 BLX             sub_4009AA8
008 00 20                       MOVS            R0, #0
008 10 BD                       POP             {R4,PC}
                ; End of function pal_MemAlloc_internal
```

Fig. 3.16. Malloc() implementation in the pal segment

Back in *cc_DecodeSetupIndMsg()* function, there is a call to a function labeled as *cc_GetNsCallTypefromNtwkCallType*. That received as first argument the Transfer Capability, as second, the bearer capability pointer, and finally the bearer capability length. See

in Figure 3.17 how is first checked if the bearer capability 1 is included, then the **length** and the pointer are stored in variables that are passed to *cc_GetNsCallTypefromNtwkCallType* as arguments. See Figure 3.17.

```
if ( cc_packet_has_bearer_1_cap_ie() == 1 )
{
  bearer_cap_1_len = (unsigned __int8)cc_get_bearer_cap_1_len();
  bearer_cap_1_ie_ptr = (_BYTE *)cc_get_bearer_cap_1_ie_ptr();
  GapiData.TransferCapability = *bearer_cap_1_ie_ptr & 7;   7 (MASK) = 0000 0111
  GapiData.CallType1 = cc_GetNsCallTypefromNtwkCallType(
                         GapiData.TransferCapability,
                         bearer_cap_1_ie_ptr,
                         bearer_cap_1_len);
  dump_bearer_1_cap_ie((int)bearer_cap_1_ie_ptr, bearer_cap_1_len);
  print_network_transfer_capability(GapiData.TransferCapability);
}
```

Fig. 3.17. Call to function *cc_get_bearer_cap_1_len()*

The transfer Capability value is defined in the octet 3 of the Bearer Capability encoding a value in three bits. See in Figure 3.17 how these 3 bytes are retrieved applying a mask with the value 7 (0000 0111) to the first Byte of the bearer capability one line before.The bearer capability length is a value that is directly passed as argument **without checking if the maximum value of 16 is exceeded**.

```
15  _buf = buf;
16  _len = len;
17  v11 = 0;
18  if ( !a1 )
19    return 1;
20  if ( a1 != 1 )
21  {
22    v6 = a1 == 2;
23    if ( a1 != 2 )
24      v6 = a1 == 3;
25    if ( v6 )
26      v5 = 3;
27    else
28      v5 = 0;
29    return v5;
30  }
31  v5 = 3;
32  _bearer_cap_ptr = pal_MemAlloc(4, len + 2, (int)"../../../HEDGE/NASL3/CC/Code/Src/cc_MtCallEstablishment.c", 3347);
33  bearer_cap_ptr = _bearer_cap_ptr;
34  if ( !_bearer_cap_ptr )
35    return v5;
36  *_bearer_cap_ptr = 4;
37  _bearer_cap_ptr[1] = _len;
38  cc_CopyBufferBytes(_bearer_cap_ptr + 2, _buf, _len);
39  bearer_id = bc_create_bearer();
```

Fig. 3.18. cc_GetNsCallTypefromNtwkCallType function

Figure 3.18 shows part of the function *cc_GetNsCallTypefromNtwkCallType*, first is checked if the first argument a1(Transfer Capability) has a different value other than 1 to return the corresponding CallType code. In this case the value is 1. A call to *pal_MemAlloc* (Figure 3.18, line 32) is made, passing as argument the length value controlled by the attacker. Note that two extra bytes are allocated to include later the IE identifier (04 for bearer capability 1) and the size of the buffer. (See Figure 3.18 lines 36 and 37 respectively).

Later, all the data is copied to a buffer(Figure 3.18, line 38). Note that a Buffer Overflow does not exist during this copy since the number of bytes copied are the same as

the amount of bytes passed to the malloc argument. However, **a buffer is being created exceeding the limits specified in the documentation**. Therefore, when this data is going to be processed by another function in the firmware, some assumptions can be made regarding the expected size of the buffer and the bug will be triggered.

Due to this vulnerability was not previously reported by any researcher performing static analysis, but it was discovered doing fuzzing to the baseband. I found it more convenient to perform some directed fuzzing on a physical device rather than looking in other tasks if allocating a buffer bigger than expected can be the source of the bug, which can be significantly harder taking into account the complexity of the firmware.

Section 3.2 presents how a package can be modified and how the device behaves when the vulnerability is triggered.

3.2. Dynamic Analysis

This section describes how to set up a BTS station emulating a cell phone provider. This way, we can interact with the phone dynamically without even touching it. This provides us two advantages: Emulating the cell phone provider, gives us the ability to send crafted messages and see how the phone reacts to them. Second, having a real phone we have access to the logs created, therefore the ability to learn more about what is happening inside the phone when a package is sent.

In particular, we will be using YateBTS to create different variations of the original SETUP package to finally craft a package modifying the bearer capability IE that crashes the baseband of our phone.

3.2.1. Environment setup

When emulating a telecommunication provider it is very important to have into account the legal aspects before setting up a BTS. See Section 4.

In order to set up our base station. We need the elements that compose the GSM infrastructure from the Mobile Station to the BTS.

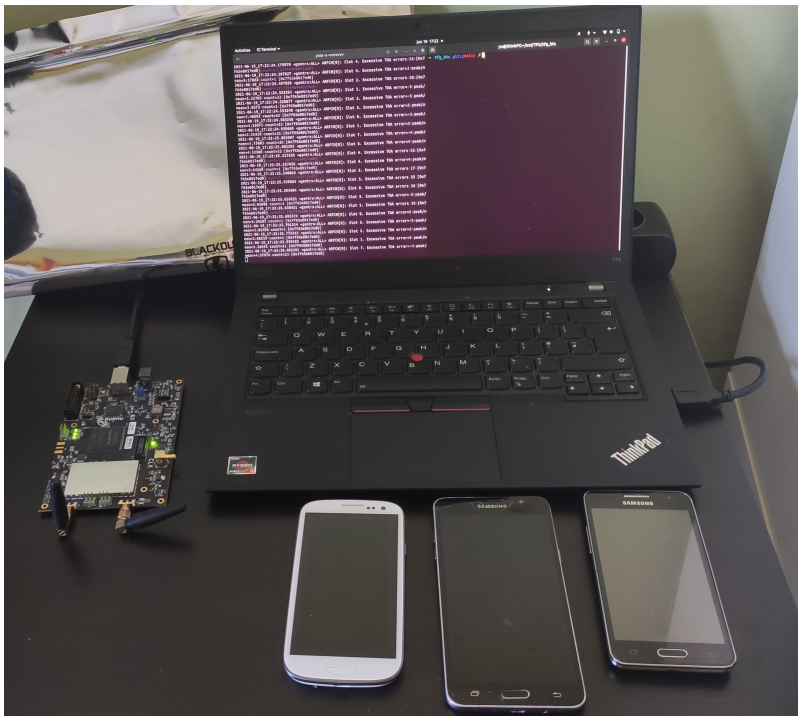


Fig. 3.19. Lab components

- **Phone:** We need the Mobile Equipment in order to carry the SIM card. The phones used in this work are: Samsung Galaxy S3, Samsung Galaxy J5 and a Samsung Galaxy J7.
- **SIM card:** Since this card will connect to our base station and not to the cell phone provider. It can be an old SIM card discontinued. No authentication is required from our base station against the SIM card since we are using GSM.
- **Software-defined radio (SDR):** This is a hardware component that transmits and receives the radio frequencies defined by a Software Engine. In this case we are using BladeRF x40 SDR.
- **Telephony software Engine:** In order to implement the GSM protocol to use along with our hardware. Essentially, this software replaces communication provider core network infrastructure from layer 3.The engine we are going to use is YateBTS.
- **Personal Computer:** Used to compile YateBTS and to communicate with the SDR using all the drivers needed and providing a communication interface with the SDR. Note that for the BladeRF x40 is needed a computer with USB 3.0.

In order to set up our lab we need to install YateBTS and configure it to work with our BladeRF x40. Refer to the official website of YateBTS to learn the prerequisites [19], install YateBTS [20] and run it [21].

Once everything is configured, we need to force our phone to use 2G (GSM). By default, modern phones will connect to the newest protocols (3G or 4G).

Once we have our phone connected to our BTS, we get a phone number by SMS. Opening a telnet terminal in the port *5038* our yateBTS we can run the following commands in order to make a call specifying the number provided earlier by the BTS.

3.2.2. Modify YateBTS packages

The code of YateBTS is written in C++ and publicly available. Therefore, it can be modified to send crafted messages.

To modify fields of a message, I located the class *yatebts/mbts/GSM/GSML2LAPDm.cpp*. All Layer 2 outgoing messages go through this class. This makes it the best place to modify our SETUP package. Modifying the code of YateBTS to send a crafted package from scratch can be done, but it is needed to take into account all the data being sent in lower layers which makes the process more complex.

Since the Setup message is sent when a call is made, it is more convenient to modify the class *GSML2LAPDm.cpp* to detect the setup message and modify it before it is sent to the Mobile Equipment. Then, it is just needed to perform a call from YateBTS and the crafted package will be sent automatically.

To implement the behaviour explained above, I added the following function that intercepts all messages and detects if the outgoing packet is a setup message. As a proof of concept I included the field *Calling party BCD Number* in the setup message. This field is used to identify the origin of the call, in other words, the number that appears in the phone when we receive a phone call. If this field is not specified, our phones will detect it as *Unknown*.

```
1  void L2LAPDm::calling_bcd_number(L3Frame& frame, unsigned char *buf,
       unsigned char *new_buf) const
2  {
3          unsigned int size=0;
4          unsigned int frame_bytes =0;
5
6          if ((buf[1] & 0x3f) == 5) {
7                  OBJLOG(NOTICE) << "Identified setup packet";
8
9                  //INSERT CALLING BCD NUMBER
```

```
                    buf[2]=0x5c; //IEI calling BCD Number
                    buf[3]=0x04; //Length field
                    buf[4]=0x81; //1(extension) 000(type of number)
                        0001(Numbering plan identification)
                    buf[5]=0x21;
                    buf[6]=0x43;
                    buf[7]=0x65;

                    size=8;
                    memcpy(new_buf,buf, size);

                    //Modify the package
                    frame.resize(size * 8);
                    frame.unpack(new_buf);
                    memset(new_buf, 0, sizeof(new_buf));
                    frame.pack(new_buf);
                    frame_bytes = frame.size() / 8;
                    OBJLOG(NOTICE) << "Frame modified. New size " <<
                        frame_bytes  << "NEW DUMP Frame: " << hexdump(
                        new_buf, frame_bytes);
            }
    }
```

Listing 3.1. Modification of the Calling BCD Number

Listing 3.1 shows the script that modifies the *Calling party BCD Number*. It works as follows: First,it is checked if it is a setup package. The first byte of the message is the protocol discriminator, and the second byte identifies the message type (See Figure 3.10). The identifier for a setup message is the value *0x05* for the lower 6 bits of this second byte. Figure 3.20.

8	7	6	5	4	3	2	1	
x	x	0	0	0	0	0	0	escape to nationally specific message types; see 1) below
x	x	0	0	-	-	-	-	Call establishment messages:
				0	0	0	1	- ALERTING
				1	0	0	0	- CALL CONFIRMED
				0	0	1	0	- CALL PROCEEDING
				0	1	1	1	- CONNECT
				1	1	1	1	- CONNECT ACKNOWLEDGE
				1	1	1	0	- EMERGENCY SETUP
				0	0	1	1	- PROGRESS
				0	1	0	0	- CC-ESTABLISHMENT
				0	1	1	0	- CC-ESTABLISHMENT CONFIRMED
				1	0	1	1	- RECALL
				1	0	0	1	- START CC
				0	1	0	1	- SETUP

Fig. 3.20. Setup message code

Since the standard just defines the 6 lower bits, a mask is required in order to isolate

these bits and then compare with the value 5.

if ((buf[1] & 0x3f) == 5)

	X	X	0	0	0	1	0	1	(**buf[1]**)
&	0	0	1	1	1	1	1	1	(**0x3f**)
	0	0	0	0	0	1	0	1	(**0x05**)

The default message is sent without any other field, therefore is needed to add the following data into the buffer. First, the value **0x5C**, that corresponds to the IE Identifier of the *Calling BCD Number*(See Figure 3.10). The rest of the buffer is filled following the message structure provided by the documentation (See Figure 3.21). The length of the following is **4 Bytes (0x04)**, the second Byte contains the Numbering plan identification, the type of number and the extension. The following values represent the value of the calling number.

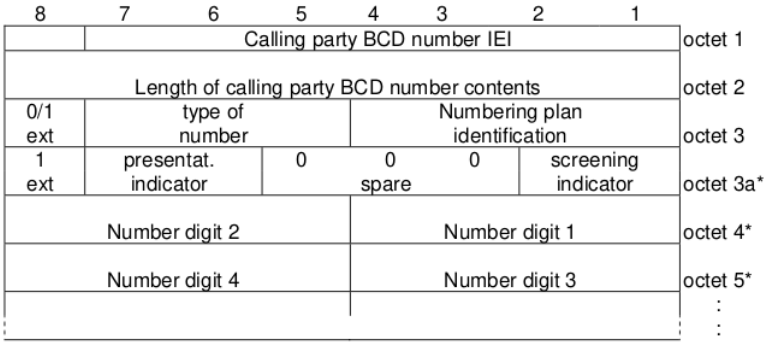


Fig. 3.21. Structure of Calling party BCD number IE

Figure 3.22 shows the phone ringing with the number specified: **21 34 65 → 12 34 56**

Fig. 3.22. Incoming call with the BCD Calling value specified in the setup message

3.2.3. Over-the-air baseband crash

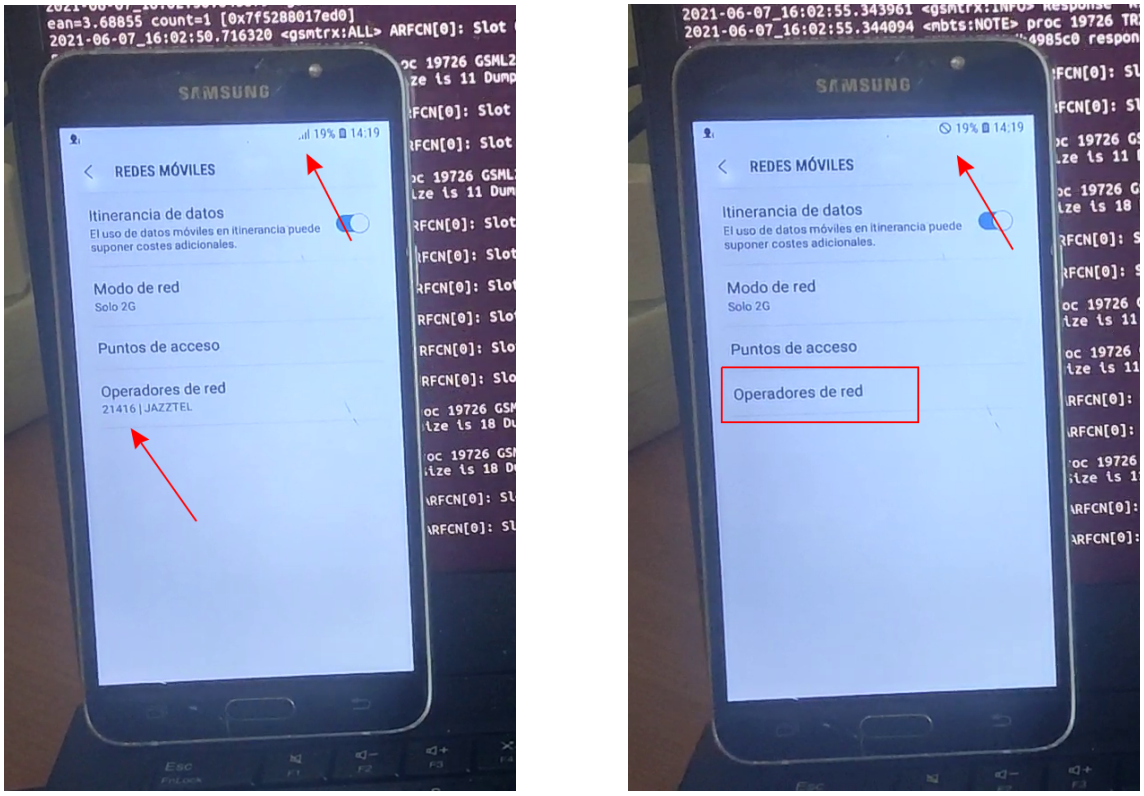
Once we managed to modify the setup message to include other fields. We modify the *Bearer Capability* field. The code is similar to the one presented in Listing 3.1, first it must be checked that the outgoing message is a Setup package. Then, we need to add the bytes that are part of this information element. In this case, we are looking for an overflow vulnerability (See Section 2.4.1), therefore, in order to exploit it, we set a length value much bigger than the expected along with the same amount of bytes specified in the length field.

```
1  int overflow=0x92; //150 Bytes
2
3  buf[2]=0x04; //IEI BEARER CAPABILITY 1
4  buf[3]=overflow; //LENGTH
5
6  memset(&buf[4], 0x07, overflow);
7
8  size=4+overflow;
9  memcpy(new_buf,buf, size);
```

Listing 3.2. Setup message with Bearer Capability buffer overflow

Listing 3.2 shows how the setup message is built. Note that the function *memset* is used

in order to fill a buffer with the value *0x07* with as many bytes as specified in the overflow variable. After trying some values and lengths, this results in the phone losing connection. See Figure 3.23



(a) Device before receiving the crafted SETUP message

(b) Device after receiving the crafted SETUP message

Fig. 3.23. Phone crash after receiving the crafted SETUP package

Having the device connected to a computer with adb(Android Debug Bridge) it is possible to run the following command to retrieve the logs that contain radio/telephony messages.

```
1  adb logcat -b radio
```

Figure 3.24 shows part of the output of these logs. Note that at the moment the device loses connection, there is a message indicating that **RILD has crashed** due to the buffer overflow in the baseband. As explained in Section 2.1. RILD is a service running in the Application Processor that manages the communication with the baseband. The description of the security patch states that this vulnerability allows arbitrary code execution. It is very likely that a pointer to a function or return address has been overwritten with an invalid memory address due to there is no validation of the buffer length when allocating the memory, making the baseband processor and the RILD service crash.

In order to achieve code execution is needed to overwrite the pointer with the memory

address containing the code we want to execute. The development of an exploit that provides code execution using this vulnerability is out of the scope of this work.

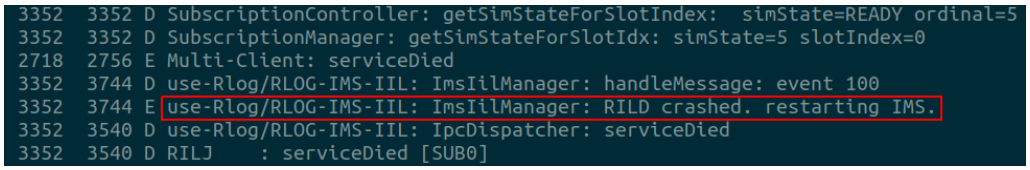


Fig. 3.24. Logcat output. Crash of RILD service

4. LEGAL FRAMEWORK

In order to perform our research and test the bug on a real phone it is necessary to set up a BTS where we can craft the correct packets to send to the phone and exploit the vulnerability. Since these messages are being transported over the air. An external phone can be connected unintentionally to our BTS instead of the legitimate cell tower. Could be the case that a user performing an emergency call is connected accidentally to our BTS.

In Spain the attribution of frequencies is regulated by "Ministerio de Energía, Turismo y Agenda Digital". In specific the current regulation is:
"Orden ETU/1033/2017, de 25 de octubre, por la que se aprueba el cuadro nacional de atribución de frecuencias".[22]

Since in most countries it is not legal to emit these waves without permission and could even interfere with our research, it is very convenient to perform our tests in a controlled environment, this could be in an isolated environment where the range of the signal emitted will not interfere with other devices. In case this is not possible, the easiest way is to purchase a Faraday Bag and put inside the items we need to perform our tests.

A Faraday bag is an enclosure that is used to shield things from electromagnetic fields, this means, no radio frequencies can go in and out from our controlled environment.

5. SOCIO-ECONOMIC FRAMEWORK

The number of phones sold in 2020 has been multiplied by 4 from 2010. This makes these devices a very interesting target for attackers. This section shows a statistical estimation of how many brand new phones are being sold to the consumer containing n-day vulnerabilities. Using this estimation model is computed the number of brand new Samsung phones carrying an Exynos chipset that were sold to end users with a known vulnerability like the one presented in Section 3. See Figure 5.1.

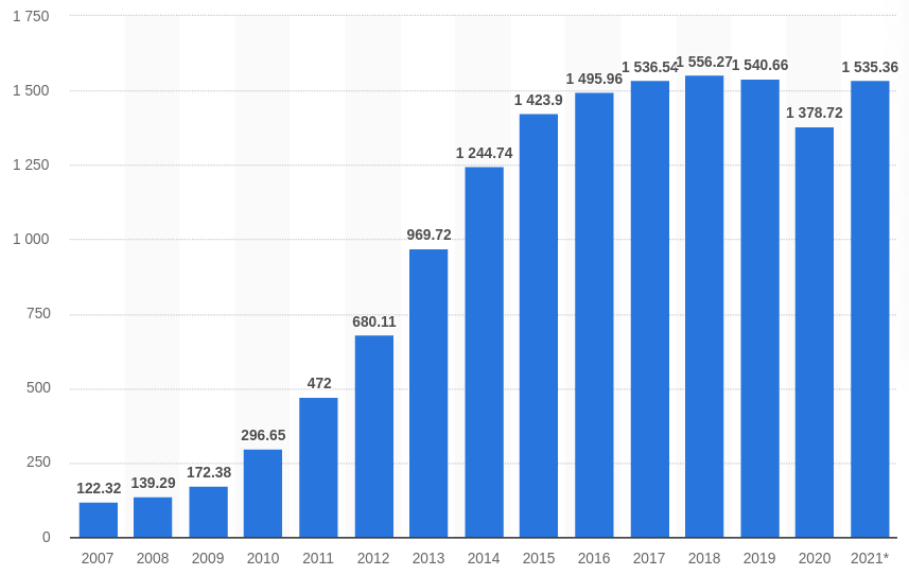


Fig. 5.1. Number of smartphones sold to end users worldwide from 2007 to 2021 [23]

At the moment a vulnerability is reported to the vendor and is fixed, all the devices manufactured from that time include a security update, so the bug can not be exploited. The affected phones already sold will receive an update via OTA(Over the air) to fix the bug. However, there will be a lot of devices that will not receive the update, hence they will remain vulnerable. There will even be brand new devices that at the moment they are purchased by the consumer they will have a **known vulnerability** that could be exploited as soon as the user inserts the SIM card and connects to a cellular network.

This is possible because phones are not sold directly from the manufacturer to the final consumer. These are purchased first by Retail Companies that store the product in Warehouses during a given period. This period is determined by the *Inventory Turnover*, (from now, IT) that describes how many times the company replaces its inventory in one year. This value is very important for retail companies in order to control the stock and satisfy the consumer needs. In most retail businesses, this value ranges from 8 to 10. [24] We get the number of phones that are replaced in each inventory by dividing the total number of phones sold in a year by TI.

Because all of the vulnerable phones in the warehouse are in a box ready to be sold at the time a vulnerability is disclosed and an update is released, all vulnerable phones will not be updated when sold.

$$Vulnerable\ phones\ sold = \frac{Potential\ vulnerable\ devices\ per\ year}{IT} \tag{5.1}$$

In 2020, 1378.72 smartphones were sold worldwide [23]. In this year, Samsung shipped 58.3(Q1), 54.2(Q2), 80.4(Q3), 73.9(Q4) million of units[25] . Which means a total of 266.8 million Samsung phones sold in 2020. However, as explained in Section 2.3 there are phones that are built with Qualcomm processors instead of Samsung Exynos processors and other non-Samsung devices, like some Meizu models that carry Exynos chipset.

Therefore is more precise to take into account that Samsung System-on-chip had a market share of 11% in 2020 [26] and 1378.72 phones were sold worldwide in that year [23], therefore, we can determine that 151.58 ($1378.72 \cdot 0.11$) millions of devices running Exynos chipset were sold worldwide in 2020. Estimating that just 60% of these Exynos devices are affected by a given Baseband vulnerability, it is estimated that 90.95 million of potential vulnerable devices were sold. Considering that *amazon.com* had a TI of 10.5 in 2020. We can use the Formula 5.1 to estimate that **8.66 million of phones** were sold as brand new containing known security flaws that can be identified exploited as is proved in this work (See Section 3).

6. BUDGET

This chapter describes all the costs involved in the development of this work.

6.1. Direct costs

This section reflects the costs needed to implement the research carried on in this work. It is considered the cost of the hardware, software and human resources.

Hardware
During the development of this work, two laptops have been used. A primary one to perform the tasks of research and static analysis and a secondary laptop to connect to the bladeRF and work as a BTS. Note that a low cost computer, like a raspberry pi or even the primary laptop can be used instead. 3 testing phones a faraday bag the Software Defined Radio, BladeRFx40 have been used to perform the dynamic analysis explained in Section 3.2.

Table 6.1 shows the characteristics of each of the hardware elements used. Note that all the prices include VAT(Value Added Tax). The cost of each of the products is computed taking into account the price as well as the used and deprecation time with the following formula.

$$Total = Price \cdot \frac{Use}{Deprecation}$$

Software and Licenses

Product	Price (€)	Use(Months)	Deprecation(Months)	Total (€)
Laptop	1200	7	72	116.67
BladeRFx40	420	4	48	35.00
Faraday bag	25	4	60	1.67
Testing phones	-	-	-	-
Galaxy A30	200	4	24	33.33
Galaxy J5	87	4	12	29.00
Galaxy S3	90	4	12	30.00
				245.67

TABLE 6.1. HARDWARE BUDGET

Software products such as the operating system(Ubuntu), YateBTS and Vistual Studio Code among others had a cost of 0€. However IDA Pro and the ARM compilers are proprietary software that needs to be purchased. A Sammobile premium account is recommended in order to download the oldest firmwares. Table 6.2

Product	License(Months)	Total (€)
Sammobile	12.00	32.50
IDA Pro	Lifetime	1,551.00
ARM32 Decompiler for IDA	Lifetime	2,171.00
Free programs	-	0.00
		3,754.50

TABLE 6.2. SOFTWARE AND LICENSES BUDGET

Human resources

There was only one person involved in the development of this work. In order to do an estimation of the human resources costs, we need to take into account the experience of the researcher and if it was provided specialized training. The more experienced the researcher, the less time will be needed to perform the work, but the cost per hour increases. In the same way, if training is provided, the time is expected to be reduced. Specialized trainings in baseband Exploitation are priced around **4000€** per person, for a 4-5 days period.[27][28].

The average cost per hour of a Junior Researcher in Spain is **13.99 €** [29]. The estimated total hours are **916** including learning, research and lab setup.

Role	Cost(€/h)	Hours	Total (€)
Researcher	13.99	916.00	12814.84
			12,814.84

TABLE 6.3. HUMAN RESOURCES TOTAL COST

6.2. Project summary

Indirect costs are 12% of the direct costs. Table 6.4 shows the total costs involved in this project. Because of the nature of this project, it involves a **High risk** of the investment, since it is not guaranteed that the vulnerability will be found.

Description	Total (€)
Hardware	245.67
Software and licenses	3,754.50
Human resources	12,814.84
Indirect costs	2,017.80
	18,832.81

TABLE 6.4. TOTAL PROJECT COSTS

7. CONCLUSION AND RESULTS

In this investigation two n-day vulnerabilities have been successfully identified in Samsung's baseband processor using reversing engineering. The first vulnerability is a Stack overflow in the function *cc_copy_cause_ie*, in charge of copying the field *cause* of an STATUS message. The second vulnerability is due to the parsing of a SETUP message with a bigger length than expected in the *Bearer Capability* field. We have proved that this vulnerability can crash the RILD service in an unpatched phone with Exynos chips using a Software-Defined-Radio and a physical device.

Furthermore, we have fulfilled our secondary objectives. It was introduced in Section 2.2 how cellular communication protocols work, the problem of no mutual authentication in GSM and how to take advantage of it. In Section 5 we have estimated that **8.66 million** of brand new phones have been sold in 2020 with this vulnerability once it was published, apart from all the discontinued devices that are still being used and not patched. These types of vulnerabilities can be exploited without even the victim realizing, putting at risk all the information in our devices.

In most modern programming languages these memory corruptions attacks are not possible, since they include protections against Buffer Overflow vulnerabilities, however these protections usually are at expense of performance and flexibility to manage memory. In Real Time Operating Systems high performance and efficiency is needed to fulfill deadlines. It is indeed the ability that provides low level languages such as C, C++ or Assembly to manage memory by the developer that yields to memory corruption vulnerabilities. It is a trade-off, if the developer needs more control over the programming language, this is more susceptible to introduce errors that can lead to a vulnerability.

The vulnerabilities described in this work affected the Second Generation of cellular communications (GSM), however, recent cellular generations like UMTS and LTE are more complex and less explored protocols that could be of interest for research. As security mechanisms emerge to protect our systems, new vulnerabilities and ways to exploit them are discovered. It is not possible to create fully secure systems, what we can do is minimize the risk. At the time these protocols were implemented in the baseband processor, Software-Defined-Radios did not exist, therefore, even if a vulnerability was found, it did not mean a security risk, since the equipment to exploit it was not affordable by most independent researchers. As proven, nowadays this is possible.

BIBLIOGRAPHY

[1] M. M. Grant Hernandez, *Reversing & emulating samsung's shannon baseband*, hardwear.io, 2020. [Online]. Available: https://www.youtube.com/watch?v=ypxgXNtvlgA.

[2] *Digital cellular telecommunications system (phase 2+) (gsm)*, V13.7.0, ETSI TS 124 008, Oct. 2016.

[3] N. Boudriga, *Security of mobile communications*, eng. Boca Raton (Florida): : Auerbach, 2010.

[4] G. Cattaneo, G. Maio, and U. Petrillo, "Security issues and attacks on the gsm standard: A review," *JOURNAL OF UNIVERSAL COMPUTER SCIENCE*, vol. 19, pp. 2437–2452, Jan. 2013. DOI: 10.3217/jucs-019-16-2437.

[5] *Arm cortex-r7 mpcore technical reference manual*. [Online]. Available: https://developer.arm.com/documentation/ddi0458/d/.

[6] *Arm cortex-r series programmer's guide*. [Online]. Available: https://developer.arm.com/documentation/den0042/a/.

[7] *Android*. [Online]. Available: https://www.android.com/.

[8] *Linux kernel*. [Online]. Available: https://www.kernel.org/.

[9] *Sammobile*. [Online]. Available: https://www.sammobile.com/.

[10] Y. Younan, W. Joosen, and F. Piessens, "Code injection in c and c++ : A survey of vulnerabilities and countermeasures," Jun. 2021.

[11] D. K. Nico Golde, *Breaking band*, REcon, 2016. [Online]. Available: https://www.youtube.com/watch?v=o280NiZjNu8 (visited on 02/07/2021).

[12] D. K. Nico Golde, *Breaking band: Reverse engineering and exploiting the shannon baseband*, Comsecuris, 2016. [Online]. Available: https://comsecuris.com/slides/recon2016-breaking_band.pdf.

[13] *Cve-2015-8546 detail*, NVD. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2015-8546.

[14] A. Cama, *A walk with shannon: A walkthrough of a pwn2own baseband exploit*, Insomni'Hack, 2018. [Online]. Available: https://www.youtube.com/watch?v=6bpxrfB9ioo (visited on 03/24/2021).

[15] grant-h, "Shannonee dynamic analysis environment," unpublished, N.D, [Online]. Available: https://github.com/grant-h/ShannonEE.

[16] T. N. Jesse Hertz, *Triforceafl*, Github. [Online]. Available: https://github.com/nccgroup/TriforceAFL.

[17] Comsecuris, *Helpful scripts for reversing engineering shannon baseband*, Github. [Online]. Available: https://github.com/Comsecuris/shannonRE.

[18] A. Cama, *A look at the samsung shannon baseband*, Keen Security Lab, 2018. [Online]. Available: https://www.youtube.com/watch?v=b5U4tG243-o (visited on 03/24/2021).

[19] *Install prerequisites for yatebts*, YateBTS. [Online]. Available: https://wiki.yatebts.com/index.php/Prerequisites.

[20] *Install yatebts*, YateBTS. [Online]. Available: https://wiki.yatebts.com/index.php/Installing.

[21] *Run yatebts*, YateBTS. [Online]. Available: https://wiki.yatebts.com/index.php/Running.

[22] Ministerio de, Energía, Turismo y Agenda Digital. (Oct. 2017), [Online]. Available: https://www.boe.es/eli/es/o/2017/10/25/etu1033/dof/spa/pdf.

[23] *Number of smartphones sold to end users worldwide from 2007 to 2021*, Statistia, 2021. [Online]. Available: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/ (visited on 02/07/2021).

[24] *Inventory turnover*, wikipedia.rog, 2020. [Online]. Available: https://es.wikipedia.org/wiki/Rotaci%C3%B3n_del_inventario (visited on 02/07/2021).

[25] *Smartphone unit shipments of samsung worldwide by quarter from first quarter 2010 to first quarter 2021*, Statistia, 2021. [Online]. Available: https://www.statista.com/statistics/299144/samsung-smartphone-shipments-worldwide/ (visited on 02/07/2021).

[26] *Smartphone application processor (ap)/system-on-chip (soc) vendor shipment share worldwide in 2020 and 2021*, Statistia, 2021. [Online]. Available: https://www.statista.com/statistics/796887/smartphone-system-on-chip-market-share-by-vendor-worldwide/ (visited on 02/07/2021).

[27] N. Artenstein, *Practical baseband exploitation training*, ringzer0. [Online]. Available: https://www.ringzer0.training/2020-august/practical-baseband-exploitation.html.

[28] A. Cama, *Baseband exploitation training*, offensivecon. [Online]. Available: https://www.offensivecon.org/trainings/2020/baseband.html.

[29] *Salary junior researcher in spain*, Glassdoor. [Online]. Available: https://www.glassdoor.es/Salaries/spain-junior-researcher-salary-SRCH_IL.0,5_IN219_KO6,23.htm?countryRedirect=true.

[30] *GSM architecture, protocols and services*, eng, 3rd ed., English lang. ed. Chichester, U.K.: Wiley, 2009.

[31] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *10th USENIX Security Symposium*, 2001.

[32] S. Soim, A. S. Handayani, A. Taqwa, N. Shadrina, *et al.*, "The implementation of yatebts based gsm using raspberry pi," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1500, 2020, p. 012 005.

[33] O. Dunkelman, N. Keller, and A. Shamir, "A practical-time attack on the a5/3 cryptosystem used in third generation gsm telephony.," *IACR Cryptol. ePrint Arch.*, vol. 2010, p. 13, 2010.

[34] *Yatebts*. [Online]. Available: https://yatebts.com/.

[35] *Global smartphone application processor (ap) market share: By quarter*, Counterpoint, 2021. [Online]. Available: https://www.counterpointresearch.com/global-smartphone-ap-market-share/ (visited on 02/07/2021).

[36] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, Califor-nia, USA, vol. 41, 2005, p. 46.

[37] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[38] D. Maier, L. Seidel, and S. Park, "Basesafe: Baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020, pp. 122–132.

[39] *Inventory turnover for amazon.com, inc.* finbox.xom, 2020. [Online]. Available: https://finbox.com/NASDAQGS:AMZN/explorer/inventory_turnover (visited on 02/07/2021).

[40] *Zerodium exploit acquisition program*, Zerodium, 2021. [Online]. Available: https://zerodium.com/program.html (visited on 02/07/2021).

[41] *Buffer overflow attack*, Imperva. [Online]. Available: https://www.imperva.com/learn/application-security/buffer-overflow/ (visited on 02/07/2021).

[42] V. F. David Berard, "How to design a baseband debugger," [Online]. Available: https://www.sstic.org/media/SSTIC2020/SSTIC-actes/how_to_design_a_baseband_debugger/SSTIC2020-Article-how_to_design_a_baseband_debugger-berard_fargues.pdf.

[43] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, Nov. 1996. [Online]. Available: http://www.phrack.com/issues.html?issue=49&id=14.