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

Ralf-Philipp Weinmann
University of Luxembourg
<ralf-philipp.weinmann@uni.lu>

Abstract

Published attacks against smartphones have concentrated on software running on the application processor. With numerous countermeasures like ASLR, DEP and code signing being deployed by operating system vendors, practical exploitation of memory corruptions on this processor has become a time-consuming endeavor. At the same time, the cellular baseband stack of most smartphones runs on a separate processor and is significantly less hardened, if at all.

In this paper we demonstrate the risk of remotely exploitable memory corruptions in cellular baseband stacks. We analyze two widely deployed baseband stacks and give exemplary cases of memory corruptions that can be leveraged to inject and execute arbitrary code on the baseband processor. The vulnerabilities can be triggered over the air interface using a rogue GSM base station, for instance using OpenBTS together with a USRP software defined radio.

Keywords: baseband security; radio firmware; memory corruption; GSM

1 Introduction

Despite recent deployments of 4G networks, Global System for Mobile Communications (GSM) [7] still is the prevalent standard for cellular communications. With billions of GSM handsets deployed, about 70% of all cellular connections in 2011 were estimated to have been performed using GSM as bearer technology[1]. Moreover, GSM will not go away soon, as even the majority of Long Term Evolution (LTE) devices are backwards-compatible not only with 3G technology but with GSM to provide connectivity in areas lacking both 4G and 3G coverage.

[1] according to market data by Wireless Intelligence

While the cryptographic algorithms A5/1 and A5/2 used for link-level encryption of voice data in GSM have been practically broken [2, 1, 13] and interception attacks have been shown to be easily possible [20, 21, 14] with off-the-shelf hardware, only little effort has been directed at researching the security of the software directly interfacing with the cellular network, the so-called cellular baseband stack.

In the past, spoofing a GSM network required a significant investment, which limited the set of possible attackers. When GSM radio stacks were implemented, attacks against end devices were not much of a concern. Hence checks on messages arriving over the air interface were lax as long as the stack passed interoperability tests and certifications. Open-source solutions such as OpenBTS [4] allow anyone to run their own GSM network at a fraction of the cost of carrier-grade equipment, using a simple and cheap software-defined radio. This development has made GSM security explorations possible for a significantly larger set of security researchers. Indeed, as the reader will see in the following, insufficient verification of input parameters transmitted over the air interface can lead to remotely exploitable memory corruptions in the baseband stack.

Let us briefly describe our attack scenario: The attacker will operate a rogue Base Transceiver Station (BTS) in vicinity to the targeted Mobile Station (MS). The rogue BTS sends out system information messages announcing the availability of a network that the targeted mobile station is willing to connect to. As the primary criterion for network reception is signal strength, the attacker can force the MS to connect to its rogue base station by simply transmitting with a stronger signal than the legitimate base station. This will not happen instantaneously, but the process can be sped up by using a GSM jammer to selectively jam the frequency of the legitimate BTS. This scenario is very similar to the one used by IMSI catchers. Since GSM does not provide mutual authentication, there is no protection against fake BTSs.

Mobile stations come in different types: examples are USB data dongles providing connectivity to laptops, tablet devices with cellular connectivity and last but not least, cellular phones. The class of cellular phones can be divided into two types: so-called "feature phones" which only offer their users basic functionalities such as making and receiving calls and sending and receiving text messages and "smartphones" which can be considered as personal assistants. Smartphones allow their users to perform a wide variety of tasks; such as browsing the Web, sending and receiving email, installing custom applications, taking pictures, shooting video, etc. Although the results described in this paper may apply to *all handsets* running vulnerable protocol stacks, we made a deliberate decision to focus our research on smartphones, as they are the most interesting targets for real-world attackers. Besides storing valuable personal data, smartphones have become the gateway to the digital world for many people. In 2011, the number of smartphones shipped surpassed the number of personal PCs and tablet PCs combined.

A paper describing the anatomy of modern GSM telephones has been written by Welte [24]. Although the line drawn between smartphones and feature phones is fluid and shifting, we can use the following distinction in their hardware to separate the two: Feature phones only have a single CPU that runs an operating system that both displays the user interface and at the same time runs the baseband software stack. On the other hand, the majority of modern smartphones contain at least two CPUs[2], the *application processor*, which handles the user interface and runs the applications installed by the user and a second CPU, the *baseband processor*, that handles connectivity to the cellular network. Some smartphone designs use a shared-memory architecture where the baseband processor can access all of the application processor's memory space while other designs have better isolation, i.e. the baseband processor and the application processor have separate memories and exchange messages through dedicated communication channel, e.g. a serial line or a small shared-memory segment (see Figure 1).

Publicly demonstrated attacks against smartphones have concentrated on exploiting vulnerabilities in software running on the application processor.

Specialized knowledge and experience is required to implement standards that are specified across several hundred documents; this is why baseband chip vendors usually sell their chips together with the corresponding software to drive it; this piece of software is called the "baseband software stack". Companies that currently sell GSM/3G baseband chips and stacks are: Qualcomm, Intel (formerly Infineon), Broadcom, Texas Instruments, ST-Ericsson, Renesas (formerly Nokia), Marvell, MediaTek, NVidia (formerly Icera), VIA Telecom and Spreadtrum.

[2]which may or may not be on the same die

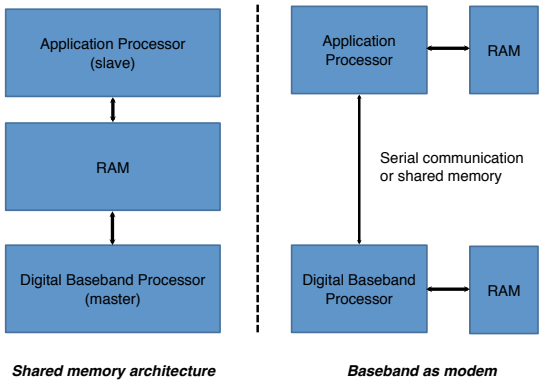

Figure 1: Common architectures employed in smartphone designs

According to a recent market report published by Strategy Analytics [15], Qualcomm and Intel (using the Comneon stack) together captured 60% of the baseband revenue in 2011, hence we have contentracted our research on cellphones using these chips.

Baseband software stacks however are less hardened against attacks than the code running on the application CPU; this can be witnessed by examining so-called "software unlocks" written for circumventing the network locks of Apple's iPhone in a non-permanent manner [16]. These are local exploits executed with each start-up of the telephone. They work by the application processor sending a sequence of AT commands to the baseband. This sequence triggers a memory corruption vulnerability in the AT command interpreter of the baseband stack. Most of the vulnerabilities exploited so far have been stack buffer overflows.

Our contribution We analyze which areas of GSM baseband stacks most likely contain programming errors leading to remotely exploitable memory corruptions. To drive our point home, we describe two bugs for two vendors with a high market share that we found during our research. We discuss what difficulties had to be overcome to exploit these memory corruptions and what the resulting impact is.

Related Work Mulliner, Golde and Seifert [18] systematically analyzed the resilience of a number of mobile phones against malformed short messages using fuzzing and demonstrated numerous remotely exploitable denial of service attacks using this vector – yet it is unclear whether any of the described vulnerabilities lead to re-

mote code execution. At Black Hat 2009 Miller and Mulliner presented a vulnerability in the SMS parsing functionality of iPhone [17] that can lead to remote code execution; this attack does not require user interaction, but it exploits a bug in `CommCenter`, which is running on the application processor.

Structure of the paper The paper is organized as follows: Section 2 provides some background and describes the relevant aspects of GSM. Section 3 describes how we performed our vulnerability analysis. Section 4 investigates the difficulty of leveraging such vulnerabilities into remote code execution. Section 5 gives an impact assessment of our research and Section 6 concludes this paper, giving an outlook where future research in this area is headed.

2 Background

Until our first presentations in 2010, no one had demonstrated an attack resulting in remote code execution in a baseband stack. This is moderately surprising. By fuzzing handsets, many crashes in the baseband stacks can be found quickly. However most of these crashes seem to not be triggered by memory corruptions. To separate the wheat from the chaff and to leverage the inputs that indeed cause memory corruptions, a deeper understanding of the baseband stack is necessary. This seems to have been the primary reason hindering security researchers from making progress in this area.

2.1 Local exploits and unlocks

For years, the primary incentive driving the reverse-engineering of cellphone firmwares has been the "unlock scene". The existence of this scene is owed to the lock-in model many network operators employ. A so-called "network lock" causes a handset to only accept SIM cards of the operator selling the handset whereas a SIM lock ties the handset to a specific SIM card (either identified by the IMSI or the ICCID). Implementing one of these two restrictions – which are implemented in the baseband firmware – allows a carrier to sell the locked handset for a cheaper, subsidized price. These barriers have been circumvented in a number of ways: missing integrity checks in bootloaders, broken integrity verification routines for flashing firmware updates and other logic errors were exploited. However, more recently memory corruption vulnerabilities (mostly stack buffer overflows) in AT command parsing routines and the SIM Toolkit functionality have been used to perform unlocks for the iPhone and Windows Mobile phones produced by HTC.

2.2 GSM layers and information elements

The layering of cellular protocols does not cleanly map to the OSI model. The GSM protocol stack on the MS consists of several layers (see Figure 2, adapted from [7]) of which only the lowest three are considered in this paper. The physical layer (layer 1) of the air interface uses Gaussian Minimum Shift Keying (GMSK) for modulating binary sequences and a combination of Time Division Multiple Access (TDMA), Frequency Division Multiple Access (FDMA) and frequency hopping for transmitting frames. The physical layer also implements logical signalling channels.

The data-link layer (layer 2) uses Link Access Procedure on Dm Channels (LAPDm) which is a simplified version of ISDN's Link Access Protocol Channel D (LAPD) that has been adapted to the air interface. LAPDm handles transport of messages between protocol entities of layer 3 and signaling tasks.

Layer 3 is significantly more complex and can be subdivided into the following sublayers (ordered from bottom to top of the stack again):

- Radio Resource Management (RR): e.g. channel set-up and tear-down
- Mobility Management (MM): e.g. location updates
- Connection Management (CM): call control (call establishment/release), supplementary services (e.g. USSD), SMS

The message format of layer 3 messages is specified in GSM 04.07 [8], the actual messages that can be exchanged are defined in GSM 04.08 [9]. A layer 3 message is composed of

- Transaction identifier or skip indicator (4 bits)
- Protocol discriminator (4 bits)
- Message type (8 bits)
- Other information elements (potentially variable length)

Of interest here are Information Elements (IEs), which come in several flavors: V, LV, T, TV and TLV where **T** denotes tag, **L** denotes length and **V** denotes value. The types V, T and TV are for Information Elements of fixed length, whereas LV and TLV are used for information elements that have varying length.

3 Vulnerability analysis

Cellular baseband stacks generally are not available in source form for non-licensees[3]. However, in 2004 the

[3] OsmoComBB, a project building an open-source GSM baseband stack for Calypso chipsets, being the exception here.

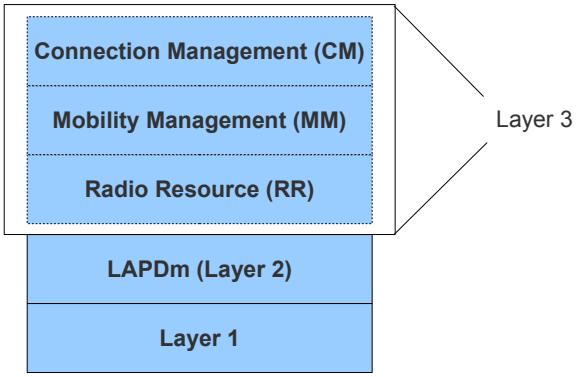

Figure 2: Layers of a GSM software stack running on a Mobile Station

source code tree for the Vitelcom TSM30 mobile phone was uploaded to a Sourceforge project [19]. Eventually it was removed, but only after having been available for download for a number of years. This source code included an old version of the Condat GSM baseband stack[4] and allowed us to obtain a general understanding of the structure of cellular stacks. This knowledge proved to be very helpful for binary analysis of our target firmwares, even though these stack were written by completely different companies on different real-time operating system.

When we began our analysis, we chose two example targets, namely the Apple iPhone 4 (using a Comneon stack on an Intel chip) and the HTC Dream (using a Qualcomm stack and chip). As baseband binaries are fairly large (multiple megabytes), we only reverse-engineered the parts that we deemed interesting. For almost all smartphones, firmware updates are readily available either from the respective website of the respective vendor or from the website of a carrier. Most of the time these updates do not only contain bug fixes or enhancements for the code running on the application CPU, but also include a full baseband binary blob. To extract this firmware image, it is necessary unpack – and for older iPhones to decrypt – the firmware update. Tools and instructions on how to do this can usually be found on message boards dedicated to tinkering with the firmware of cellular phones. The reverse-engineering process is significantly simplified by error strings and file names of source code files embedded in the binaries. These apparently are used for diagnosis on production devices using vendor applications such as Qualcomm's QXDM and Comneon's Mobile Analyzer.

Almost all baseband processors are ARM processors and therefore well supported by the IDA Pro disassembler. Moreover, Hex-Rays, the company producing IDA Pro ships a decompiler for the ARM architecture that can give reverse-engineers a significant speedup in analyzing larger codebases such as baseband firmware images.

For the iPhone, a thriving scene exists that has already reverse-engineered parts of its baseband software to create "software unlocks", software that is injected through a local baseband exploit to allow to circumvent network locks imposed by the carriers. Some documentation is on these unlocks available in a wiki [5], a detailed description of the reverse-engineering of ultrasn0w is given in [16].

Our main tools for identifying "interesting" code paths were IDA Pro and Google BinDiff. BinDiff allows us to re-identify known functions in binaries. By computing a number of metrics on the flow graphs of a function, a function "fingerprint" is obtained. A metric on the function fingerprints then allows to identify "similar" functions in other binaries. "Symbol porting", process of function re-identification, then works by observing that most functions observed in the wild are equivalent when their similarity value is high enough.

We used BinDiff to identify functions such as `memcpy()`, `memmove()` and `bcopy()` and RTOS system functions by using BinDiff to port symbols from several standard compiler libraries and RTOS binaries with symbols. This then allowed us to identify functions that used variable-length memory copies, enabling us to quickly see which of them employed insufficient length checking for the data copied. In principle, IDA Pro's FLIRT signatures can be used for this purpose as well, but they are less flexible, since they don't work on the flow graph abstraction model. For the iPhone 4 it proved to be very fruitful to start by analyzing the first generation iPhone (iPhone 2G) and identify security problems here – the work can later be ported over using BinDiff, a tool described below. The advantage of this approach is that one has to deal with a significantly reduced amount of functions since the iPhone 2G lacks UMTS and GPS functionality.

3.1 Areas of interest

Below layer 3, there usually is little potential for exploitable memory corruptions, as the messages transmitted are too short. An exception to this rule are voice codecs. Reading parts of the GSM layer 3 specification [9], an inclined reader finds multiple areas that look promising for exploitation: there are a large number of messages that specify IEs to be encoded as TLV or LV, even though in the message description it becomes clear that the values transmitted are of predefined length. This can usually lead to a mismatch between different imple-

[4]Condat was later bought by Texas Instruments. The stack can still be found in their products.

menters and hence to the most classic of all memory corruptions, the buffer overflow. At one point there may be a fixed-length buffer allocated, while at another point the message is copied into the buffer without making sure that the message indeed is smaller than the buffer, trusting the length argument specified in the information element.

Similarly, many explicit state machines exist in [9], many of which have state transitions depending on timer expiries. As some of the state transitions involve allocation and deallocation of dynamic memory, dangling pointers can arise if implementers are not carefully covering edge cases.

3.2 Classification of bugs found

To get a better understanding of which parts of the cellular baseband code are most worthwhile auditing, it makes sense to classify the (memory corruption) vulnerabilities we have found by type:

Insufficient length checks:
These were by far the most common types of bugs we encountered. They usually resulted in data on the heap or on the stack being overwritten, which can be leveraged by an attacker to gain control over the execution flow using the usual methods. Exploitation of heap corruption bugs is vastly more easy than on current desktop platforms in these embedded systems – initially, we didn't find a single example of safe unlinking being employed in heap implementations of baseband operating systems[5].

Object/structure lifecycle issues
Due to the generous use of state machines in GSM, memory corruptions can arise out of lifecycle issues. These can be use-after-free bugs (e.g. a dangling pointer to a structure that has been deallocated already) or uninitialized variables (most useful when on the stack). Exploiting these issues can be more difficult than the first kind, but given the fact that there are no exploitation countermeasures in place, they shouldn't be easily discounted as "unexploitable". Common examples of state machines are the state machine used for handling incoming SMSes and for Cell Broadcasts.

Integer overflows/underflows
We only found a very small number of these bugs compared to the other kinds. This may be either because our methods for detecting them (in binary code) are not enough or because cellular code base is inherently less prone to these types of bugs

[5] in reaction to our results, at least the Qualcomm modem heap now uses safe-unlinking, though [16].

Memory information leaks
Whilst this class of vulnerabilities is not a memory corruption issue, memory information leaks can be highly useful to leverage memory corruptions better. Usually, they arise in the same context as the lifecycle issues mentioned above, at least in baseband stacks.

We did not find any format string issues; this is not surprising, given that most uses of `sprintf()`-like functions are in the diagnostic code and do not allow for arbitrary format strings to be passed.

3.3 Finding insufficient length checks

The majority of bugs found where memory corruptions that occurred to due insufficient checks on length fields being performed. In principle, these can be found by fuzzing all information elements with length fields. Fuzzing however is a very crude method. Instead the method we employed was to look at the source code for the baseband stack of the Vitel TSM30 and see which types of memory corruption problems were widespread in this code base. By identifying `memcpy()`, `memmove()` and similar memory-transfer library functions which are called with a non-constant length parameter, we identified potentially vulnerable routines and checked them individually in other baseband stacks.

3.4 Issues with dual-mode

In baseband stacks that support both GSM and UMTS, code paths between the two are often shared. This means that in some instances a code path that should only be reachable when the device is talking to a UMTS base station also is accessible using well-crafted GSM layer 3 messages (which of course are undefined in GSM 04.08). An example of this is given in the next subsection. In other cases this type of bug can lead to uninitialized variables and object lifecycle issues.

3.5 Examples of exploitable bugs found

- During the registration phase, a TMSI is assigned to the handset if it has not been seen before. This TMSI is supposed to be always 32 bits long, but a variable length field is used. Indeed, sending a longer TMSI (e.g. 128 bytes) caused the baseband stack of iPhones with the Intel/Comneon to crash.
- For authentication, the base station transmits a challenge to the handset. In GSM, this challenge is a 16 byte value called RAND. For UMTS a so-called AUTN challenge [11] is used – which is encoded as a variable length IE, but in fact is specified to also

be be 16 bytes long. Interestingly, we were able to force the Qualcomm stack to accept this variable length AUTN challenge even in a GSM layer 3 message by setting the message type to the one used for UMTS RANDs. This causes a classic stack overflow (for more than 48 bytes of AUTN), as this challenge is copied to a buffer on the stack that apparently has only been provisioned for 16 byte challenges.

The above bugs are just two of many that were found; more can easily be found by looking at all of the variable length information elements, sending long messages and subsequently locating the corresponding functions parsing them in the baseband firmware if the baseband crashes. Alternatively, one can locate all functions that copy memory and statically analyze where they are being called from – most firmware images contain debug information that allows an attacker to figure this out using the binary firmware images only. We will not list all of the bugs that we have found in this paper as we yet have to disclose them to the vendors; it currently is unclear how long it takes to get them fixed. However, bear in mind that the above examples barely scratch the surface.

4 From bugs to exploits

Developing exploits for embedded systems can be challenging if the platform is only partially understood, as is usually the case with large reverse-engineered code bases such as cellular protocol stacks. Moreover, debugging capabilities can be very primitive if JTAG access to the chip has been disabled.

However, to demonstrate the exploitability of a vulnerability, it is sufficient to make the phone perform an unexpected action: We have chosen to use the auto-answer functionality – defined in GSM specification 07.07 (*AT command set for GSM Mobile Equipment*) – which makes the phone automatically pick up an incoming call without user interaction after a predefined number of rings.

The auto-answer feature is mandatory for cellular phones and enabled by sending the command `ATS0=`n over the AT command interface to the baseband; n indicates the number of rings after which the call should be automatically picked up with $n = 0$ disabling the functionality. The above V.25ter command is a relict of the days of PSTN modems; the register `S0` was used to set the number of rings after which a modem would pick up.

To enable the auto-answer in our exploit, we first locate the AT command handler for setting the `S0` register. For stack-buffer overflows or other exploits that give us control over the program counter directly, we then load the value 1 into register `R0` and redirect the execution flow into this function. Depending on whether this setting is in RAM or whether it is backed by an EEPROM, we either need to make sure that we continue the execution correctly or we can crash without any penalties.

For heap buffer overflows that result in the attacker being able to overwrite an arbitrary location in memory[6] it may be easier to directly overwrite the location that is set in the AT command handler for `S0` instead of redirecting the execution flow. Alternatively, sometimes heap buffer overflows are followed by memory copies into a stack buffer, in which case a heap buffer overflow can propagate and trigger a stack buffer overflow.

Figure 3: Test setup: USRPv1, built-in FA-SY 1 module, laptop running OpenBTS, test phones (Motorola Backflip, Apple iPhone 2G, HTC Dream, BlackBerry Bold 9700)

4.1 Our setup

To verify our research, we used a modified Ettus Research USRPv1 together with 2 RFX-900 daughterboards. Since the clock signal of the USRPv1 is imprecise (a clock drift of 20ppm is usual) and its standard reference clock of 64Mhz less suitable than a 52Mhz clock for GSM (the GSM symbol rate is derived from a 13MHz clock), we have modified the USRP to use an an external clock and feed it clock signal produced by ClockTamer module. The USRP is connected to a Thinkpad X60 with a Core Duo CPU @1.6GHz that runs OpenBTS 2.6, modified with patches to perform the exploits listed below. Figure 3 shows a picture our setup [7].

In our tests we did not spoof a carrier but rather operated a test network with MCC 001 and MNC 01 on a frequency for which we had obtained authorization from the local regulation authorities.

[6] a so-called write 4 primitive

[7] The photo shows the first generation setup, in which we were using a FA-SY 1 module instead of a ClockTamer

4.2 Device fingerprinting

To reliably exploit vulnerabilities in baseband stacks, it is useful to identify both the device and the exact version of the stack running on the device. This can be achieved in multiple ways, the easiest of which is using the International Mobile Station Equipment Identity and Software Version Number (IMEISV) [10]. The IMEISV has the format `AA-BBBBBB-CCCCCC-DD` with decimal digits; the part `AA-BBBBBB` designates the GSM Type Approval Code (TAC), `CCCCCC` designates the serial number of the device and `DD` the software version running on the device. The IMEISV can be queried by the base station during the location update, allowing for targeted attacks. We have modified OpenBTS to query the IMEISV during registration. TACs can be mapped to the manufacturer and model name using a TAC database. The official database is maintained by TÜV SÜD BABT for the GSM Association and can not be queried by the general public. However, there exist public databases on the internet which cover a non-negligible portion of the assigned TAC space.

Alternatively, memory information leaks and minor protocol variations between baseband revisions could be used to fingerprint software versions.

4.3 Debugging baseband stacks

To gain a better understanding of the internals of a baseband stack as well as to write proof-of-concept exploits it is helpful to be able to examine memory and register contents at run time. In general, any debugging capabilities will greatly reduce the amount of development time for any exploit.

Most chipsets in mobile phones allow JTAG access to be disabled or to be access-protected, for instance with a secret key. This is done to prevent people from tampering not only with the baseband firmware but also from removing SIM locks or changing the IMEI of the phone. Whether or not JTAG is disabled usually is left up to the OEM producing the phone and not to the chipset manufacturer.

In practice, a large number of cell phones on the market do allow JTAG access as can be witnessed from the list of phones supported by dedicated cellphone repair boxes like the RIFF Box. Indeed, the HTC Dream we have chosen as an object of study for this paper does allow JTAG access to the baseband processor. However, during the boot process, JTAG access is disabled in the secondary bootloader, the OSBL. The decision to disable JTAG is made based on a flag and can be patched out by setting a breakpoint and changing the register the flag is loaded into, allowing us to have JTAG access to the baseband CPU at run time as well.

A second way to debug devices with Qualcomm stacks is through the so called DIAG interface. This is an interface usually used for device diagnostics, however it can also be used to peek and poke into memory at run time. Guillaume Delugré wrote an excellent debugger for older Qualcomm basebands [6] that has been released as open-source software. As-is this version will only work on pre-OKL4 chipsets. The HTC Dream on the other hand uses a baseband stack that runs on top of the OKL4 microkernel. This means that the DIAG task is running in ARM user mode and hence has insufficient privileges to access the needed debug registers. This situation can be remedied using a local privilege escalation in OKL4.

On Apple iPhones, JTAG access seems to be completely locked down. Hence, our debugging capabilities are limited. Baseband crash logs and baseband crash dumps are the only debugging facilities we found (an example of a baseband crash log is given in Appendix B). These are copied from the iPhone to a computer during the sync process. Alternatively crash logs can be obtained directly on jailbroken phones using an AT command, `AT+XLOG`. Baseband crash dumps can be enabled by dialing `*5005*CORE#` in the phone dialer. These can be extracted from the directory `/Library/Logs/CrashReporter/Baseband` on jailbroken phones.

4.4 Example target: HTC Dream

Turning on auto-answer on the HTC Dream turned out to be easy once we had identified the AT command function changing the S0 register. We have written an exploit for the AUTN stack buffer overflow previously described that overwrites the program counter and the register `R0` of a stack frame. The program counter with the entry point of the `S0` register handler, the register `R0` with the value 1. Since the ring counter is only stored in volatile memory, we cannot simply crash after writing this setting. Henceforth, we also needed to overwrite the program counter in the subsequent stack frame (which corresponds to the function that is the caller of the function corresponding to the stack frame we overwrote above) to make sure that execution of the thread continued normally.

To execute this exploit, no modification of the OpenBTS code base was necessary. Rather, given this single layer 3 message – less than 100 bytes long – exploitation of the AUTN bug becomes almost trivial. This payload is sent to a running OpenBTS instance to the `testcall` UDP port after establishing a channel using the OpenBTS `testcall` command. The bug will be exploited and auto-answer enabled without the user being able to notice anything.

4.5 Example target: Apple iPhone 4

Auto-answer is an undocumented feature on all iPhones that can be enabled by dialing `*5005*AANS#` in the iPhone dialer, which in turn is translated into a `ATS0=1` command by CommCenter and sent to the baseband modem device. Auto-answer is a permanent setting that is stored in non-volatile memory on the application processor, i.e. a reboot of the iPhone/crash of the baseband preserves this change.

All iPhones except the iPhone 4 CDMA and the iPhone 4S employ Intel (formerly Infineon) baseband chipsets running a Comneon stack. This stack is built on top of the ThreadX RTOS for the iPhone 4[8]. The TMSI overflow we previously described is a heap-based overflow that allows us to overwrite an arbitrary location of memory.

We have written a proof-of-concept exploit that uses malformed `LOCATION UPDATING ACCEPT` requests containing a TMSI that overwrites heap metadata of an allocation in a ThreadX memory block pool. To be able to send this to targets we had to slightly modify the OpenBTS code base to facilitate TMSIs longer than 4 bytes. A `LOCATION UPDATING REQUEST` is sent by the phone as soon as it connects to our network to which OpenBTS will send the malformed `LOCATION UPDATING ACCEPT` containing the payload. This results in auto-answer to be enabled and our phone to briefly lose connectivity to the network.

A more detailed description on how to exploit the same bug on the iPhone 2G (which also uses a Comneon stack, but running on a different RTOS), albeit in an easier way, is described in [16].

5 Impact

Successful exploitation of memory corruption in GSM baseband software stacks provides an attacker with access to privacy-relevant hardware of the telephone. Audio routing on the majority of chipsets is done on the baseband CPU, which means that it has access to the built-in microphone; similarly for built-in cameras. An attacker that has taken control over the baseband side of a telephone can monitor a user completely transparently – without visibility of the compromise from the side of the application CPU. Furthermore, given the large quantities of RAM available to the baseband on some phones, surreptitious room monitoring is possible: Simply record the audio from the microphone and store the compressed audio data to ring buffer in RAM. The payload then waits until a data connection is established and piggy-backs onto it, sending out the compressed recording to a server of its choice. A second obvious set of problems revolves around billing issues: once the attacker has control over the baseband he can place calls, send premium SMSes or cause large data transfers unbeknownst to the owner of the phone. This obviously can cause problems for both carriers and end-users. Compared to the above issues the fact that an attacker can arbitrarily and permanently brick devices by writing to regions of NVRAM that contain important device data like the IMEI looks almost like a minor problem.

The impact can be even more devastating on shared-memory designs such as the Qualcomm MSM7200 and similar platforms. On these, an ARM9 for the digital baseband and an ARM11 for the application side share the same memory, with the baseband core being the master. This means that no matter how well the operating system running on the application CPU is secured, bugs in the baseband stack with subsequent privilege escalations in OKL4 allow an attacker to take control over the whole device. In designs where the application CPU and the baseband CPU access separate memories the attacker however may still be able to elevate his access to the application CPU by exploiting bugs in one of the components interfacing the application processor with the baseband processor.

Forensics of volatile memory of the baseband stack is difficult without leveraging another exploit – protections against unlockers have made hardware forensics such as dumping RAM contents of a live chip via JTAG on most production phones hard.

6 Conclusions and Outlook

We have demonstrated that memory corruptions in baseband firmwares exist and can be practically exploited. These security problems are to be taken seriously: practical exploitation of these completely compromises the integrity of the attacked handset. Merely coming into the proximity of a malicious base station is is sufficient to take over any vulnerable handset – no user interaction is required by the bugs we have outlined above. The cost of exploitation is low enough to make these attacks a reality even for attackers with a limited budget: for the price of a mid-range laptop – USD 1500 – an attacker can buy the hardware to operate a malicious GSM cell with OpenBTS.

We have disclosed the bugs described in this paper to the affected baseband stack vendors. The TMSI overflow has been assigned the CVE identifier CVE-2010-3832 and has been fixed in the baseband firmware shipped with Apple's iOS 4.2. Although no public documentation on this matter exists, we understand that the AUTN overflow has been patched in Qualcomm's tree and updates have been sent out to the OEMs. In December 2010 we reverse-engineered an updated baseband for the HTC

[8] Nucleus PLUS is used for earlier models

Desire and confirmed that it did indeed contain a length check in the function parsing the AUTN parameter.

While we did not investigate 3G stacks in detail, we expect even handsets that operate in 3G-only mode to be vulnerable to similar memory corruption problems – even though they require mutual authentication. Femtocells with modified software allow attackers to operate rogue 3G base stations [3]. The specification of the 3GPP Radio Resource Control layer gives a significantly increased attack surface: On almost 1500 pages the most basic layer 3 protocol for 3GPP is defined [12]. Moreover, in contrast to the simple TLV encoding employed in GSM, the information elements of the RRC are ASN.1 encoded, using Packed Encoding Rules (PER). As the message parsing functions of the RRC layer can be triggered before the authentication process has completed, this gives a large attack surface.

To increase the security of baseband stacks, we suggest to vendors that baseband operating stacks undergo a systematic and continuous code audit and use hardening options similar to the ones used in desktop operating systems. This will make practical exploitation of security vulnerabilities in baseband stacks more difficult [23, 22]. Also, privilege-separation for establishing well-defined boundaries between the different portions of a baseband stack can be a very effective measure for making bugs that can be triggered by consuming untrusted data much harder to exploit; this however requires a design overhaul of the respective baseband stack.

We understand that our findings have caused extensive code reviews of multiple baseband stacks to happen.

Acknowledgements: We're grateful to Joshua Lackey and Harald Welte for providing detailed and thoughtful comments on an early draft of the paper. André Stemper (University of Luxembourg) helped in practical ways by applying his excellent soldering skills! Without the products and support of the ex-Zynamics teams, many code paths would have been much harder to analyze. Planetbeing and MuscleNerd provided invaluable tips about the iPhone 4 baseband. Last but not least, we are indebted to the WOOT reviewers for their constructive comments and to Aurélien Francillon for being an extremely kind an knowledgable shepherd to this paper.

References

[1] BARKAN, E., BIHAM, E., AND KELLER, N. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In *CRYPTO 2003* (2003), D. Boneh, Ed., vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 600–616.

[2] BIRYUKOV, A., SHAMIR, A., AND WAGNER, D. Real time cryptanalysis of A5/1 on a PC. In *FSE 2000* (2001), B. Schneier, Ed., vol. 1978 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[3] BORGAONKAR, R., GOLDE, N., AND REDON, K. Femtocells: A poisonous needle in the operators hay stack. presented at Black Hat Las Vegas 2011, July 2011.

[4] BURGESS, D. A., AND SAMRA, H. S. The Open BTS project. `http://openbts.sourceforge.net/`, Aug. 2008.

[5] COLLABORATIVE EFFORT. The iPhone Wiki. `http://theiphonewiki.com`, November 2010.

[6] DELUGRÉ, G. Rétroconception et débogage dun baseband qualcomm. In *Symposium sur la scurit des technologies de l'information et des communications (SSTIC 2012)* (June 2012), pp. 393–411.

[7] EBERSPÄCHER, J., VÖGEL, H.-J., BETTSTETTER, C., AND HARTMANN, C. *GSM – Architecture, Protocols and Services*, 3rd ed. Wiley, 2009. ISBN 0470030704.

[8] ETSI. Digital cellular telecommunications system (Phase 2+) (GSM); Mobile radio interface signalling layer 3;General aspects (GSM 04.07 version 7.3.0 Release 1998), Dec. 1999. ETSI EN 300 940 V7.7.1.

[9] ETSI. Digital cellular telecommunications system (Phase 2+) (GSM); Mobile radio interface layer 3 specification (GSM 04.08 version 7.7.1 Release 1998), Oct. 2000. ETSI EN 300 940 V7.7.1.

[10] ETSI. Digital cellular telecommunications system (Phase 2+); Numbering, addressing and identification (3GPP TS 03.03 version 7.8.0 Release 1998), Sept. 2003. ETSI TS 100 927 V7.8.0.

[11] ETSI. 3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals; Mobile radio interface Layer 3 specification; Core network protocols; Stage 3 (Release 8), Dec. 2008. 3GPP TS 24.008 V8.4.0.

[12] ETSI. Universal Mobile Telecommunications System (UMTS); Radio Resource Control (RRC); Protocol specification (3GPP TS 25.331 version 7.17.0 Release 7), July 2010. ETSI TS 125 331 V7.17.0.

[13] GÜNEYSU, T., KASPER, T., NOVOTNÝ, M., PAAR, C., AND RUPP, A. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers 57*, 11 (2008), 1498–1513.

[14] KRISSLER, S., NOHL, K., AND STEVENSON, F. A. The A5/1 security project. `http://reflextor.com/trac/a51`.

[15] KUNDOJJALA, S. Baseband market share tracker: Qualcomm and Intel together capture 60 percent of 2011 baseband revenue. `http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7261`, April 2012.

[16] MILLER, C., BLAZAKIS, D., ZOVI, D. D., ESSER, S., IOZZO, V., AND WEINMANN, R.-P. *iOS Hacker's Handbook*. Wiley, 2012, ch. 11, p. 408.

[17] MILLER, C., AND MULLINER, C. Fuzzing the phone in your phone. presented at Black Hat Las Vegas 2009, July 2009. `https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf`.

[18] MULLINER, C., GOLDE, N., AND SEIFERT, J.-P. SMS of Death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium 2011* (2011), USENIX Association.

[19] PURPLELABS. TSM30 firmware. `http://web.archive.org/web/20060627121308/http://sourceforge.net/projects/plabs`, Nov 2004. Sourceforge project has been deleted.

[20] STEVENSON, F. A. [A51] The call of Kraken. Mailing list post: `http://lists.lists.reflextor.com/pipermail/a51/2010-July/000683.html`, July 2010.

[21] THE AIRPROBE TEAM. AirProbe – an air-interface analysis tool for GSM. `http://www.airprobe.org`.

[22] THE PAX TEAM. Documentation for the PaX project: Adress Space Layout Randomization design & implementation. `http://pax.grsecurity.net/docs/aslr.txt`, Apr. 2003.

[23] THE PAX TEAM. Documentation for the PaX project: Non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`, May 2003.

[24] WELTE, H. Anatomy of contemporary GSM cellphone hardware. `http://laforge.gnumonks.org/papers/gsm_phone-anatomy-latest.pdf`, Apr. 2010.

A Example stack overflow: AUTN stack buffer overflow, Qualcomm stacks

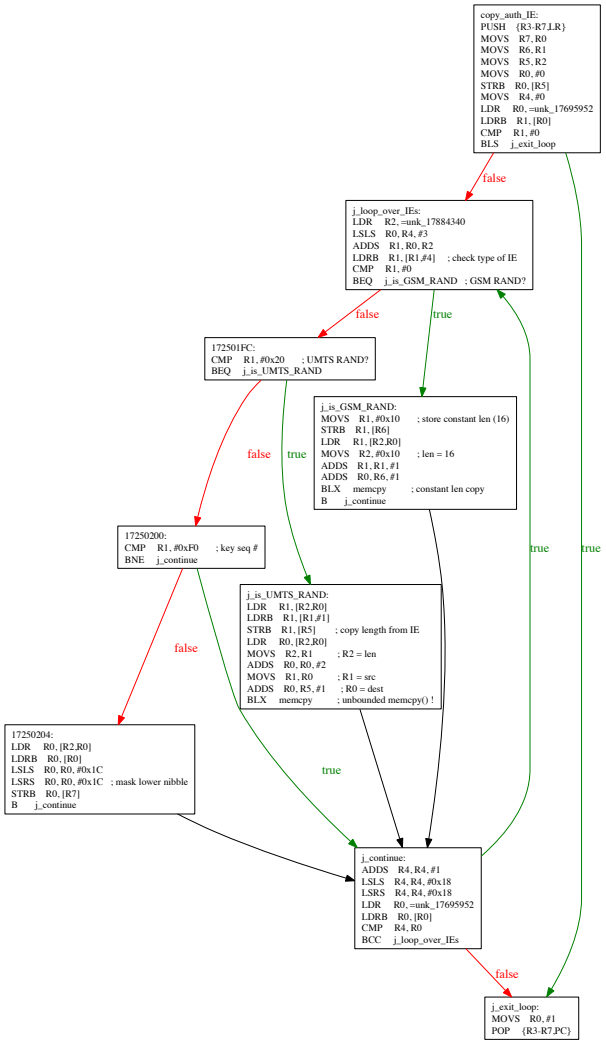

Figure 4: Disassembly of a vulnerable routine handling `AUTHENTICATE REQUEST` taken from HTC Dream radio firmware version 2.22.23.02

```
+XLOG: Exception Number: 1
Trap Class: 0xAAAA (HW DATAABORT TRAP)
Date:  06.08.2010
Time:  14:39:07
Magic: 55809
Task name: mmc:1
System Stack:
            0x73182120
            0x73183000
            0x73182120
            0x0009D0A8
            [........]

Fault registers:
DFAR: 0x73183002  DFSR: 0x00000017
IFAR: 0x00000000  IFSR: 0x00000000

Abort Mode registers:
r13:  0x0009B9C0  r14:  0x6028AA1E  SPSR: 0x00000073

System/User Mode registers:
r0:   0x73182120  r1:   0x73183000  r2:   0x73182120
r3:   0x00000070  r4:   0x0000018A  r5:   0x731829D4
r6:   0x73182120  r7:   0x00000001  r8:   0x00000000
r9:   0x00000000  r10:  0x73181000  r11:  0x00000000
r12:  0x00000000  r13:  0xFFFF3B00  r14:  0x430F295B
r15:  0x6028AA14  CPSR: 0x800001D7

FIQ Mode registers:
r8:   0x3FB98490  r9:   0x9F120729  r10:  0x49331CF4
r11:  0xCAC11D04  r12:  0xF5A4FA4A  r13:  0x60BDDE10

r14:  0x38490410  SPSR: 0x00000010

SVC Mode registers:
r13:  0x73181934  r14:  0x6028054D  SPSR: 0x20000053

IRQ Mode registers:
r13:  0xFFFF2F20  r14:  0xFFFF1104  SPSR: 0x80000053

Undefined Mode registers:
r13:  0x0009BCC0  r14:  0x6606A0A2  SPSR: 0x00000010

Secure Monitor Mode registers:
r13:  0xD986A74C  r14:  0x22883490  SPSR: 0x00000010
```

Figure 5: Baseband crash log of an iPhone 4 running baseband revision 01.59.00, triggered by a long TMSI in a `LOCATION UPDATING ACCEPT` message