kaspersky

Cinterion® EHS5 3G UMTS/HSPA Module Research

kaspersky

Table of contents

1 Abstract....4
2 Overview....5
3 Modem description....6
3.1 Modem software components....6
3.2 Types of MIDlets....6
3.3 Software update....7
3.4 Installing MIDlets on the modem....7
3.5 Debugging MIDlets and modem execution....9
4 Information security assumptions regarding the modem....12
4.1 MIDlet security....12
4.2 Modem OS security....12
5 Obtaining modem firmware....14
5.1 Building a logical image of firmware....15
5.2 Analyzing the contents of the UFS....20
6 Analysis of MIDlets security....25
6.1 FTP client security....25
6.2 Getting information about internal FS paths via a static method inside a custom MIDlet....25
6.3 Getting information about internal FS paths from a user MIDlet via a standard *FileConnector* call....26
6.4 Changing the security domain of a custom MIDlet....27
7 Modem OS security....29
7.1 Selecting interfaces for research....29
7.2 Researching modem AT commands....29
7.3 Vendor-specific AT commands....31
7.4 Fuzzing AT commands....41
7.5 SMS message processing....45
7.6 OTAP protocol analysis....52
7.7 ULP protocol analysis....54
7.8 How to read memory very, very slowly....58
7.8.1 HeapFree and HeapBase structures....59
7.8.2 Memory map....61
7.8.3 Reading memory via a heap overflow....62
7.8.4 Getting the overflow context....63

7.9 Write primitive 64
7.9.1 Free function 64
7.9.2 Thread structure 66
7.9.3 *HeapBase* structure 68
7.9.4 Strategy for handling free and occupied chunks 69
7.9.5 Selecting a new chunk 69
7.9.6 Releasing an occupied chunk 69
7.9.7 Processing fragmented SMSes 70
7.9.8 Specifics of processing each WAP message 72
7.9.9 Putting it all together 73
8 Post exploitation of network level vulnerabilities 74
8.1 Executing AT commands 74
8.2 Code execution? 76
8.3 Configuring the MMU 77
8.4 SMS FS and OTAP activation 77
9 CVE list 81
10 Conclusion 82

kaspersky

1 Abstract

Modems play an important role in enabling connectivity for a wide range of devices. This includes not only traditional mobile devices and household appliances, but also telecommunication systems in vehicles, ATMs and Automated Process Control Systems (APCS).

When integrating a modem, many product developers do not think of protecting their device from a potential modem compromise. As one of the main communication channels for the end device, the modem not only has access to the information flow between the device and the outside world, but may also have almost unlimited access to the end device's most critical systems and resources. Thus, modem security is of vital importance.

To make the problem worse, when a critical vulnerability is discovered in just one modem model and version, a significant amount of time may be required to update all the devices in which it is installed. And some of them may even not have a remote modem updating feature at all, such as a car's Telematic Control Unit (TCU). In such cases, installing the update typically requires additional effort and expense for the manufacturer of the end product to manually address each vulnerable device or vehicle.

For this reason, a particular modem manufactured by Cinterion caught our interest. When we began our assessment, the only known registered vulnerability was CVE-2020-15858[1], which is described in greater detail elsewhere[2].

We ultimately discovered several vulnerabilities, two of which are considered critical. One allows remote execution of arbitrary code at the level of the modem OS via sending specially crafted SMS messages to it. The other vulnerability allows local execution of an unsigned MIDlet with vendor privileges on the modem. When combined, the identified vulnerabilities allow an attacker to remotely obtain full control over the modem.

[1] https://www.cve.org/CVERecord?id=CVE-2020-15858

[2] https://threatpost.com/flaw-affecting-millions-iot-devices/158472/

2 Overview

Section 3 provides general information about the modem. In this section, we describe the modem’s user features, ways to interact with it according to the documentation, as well as its hardware components.

Section 5 outlines the algorithm for recovering the modem firmware from its ROM image. The recovery process involved reading the modem's NAND memory using a programmer, followed by reconstructing the logical image of the modem ROM sections from the extracted physical dump.

Section 6 focuses on evaluating the security of the MIDlets. It examines various attack vectors and outlines the potential impacts of exploiting the identified vulnerabilities.

Section 7 contains a detailed analysis of the security of the modem OS. It explains how an attacker can execute malicious code on the modem CPU by sending a few specific SMS messages.

Section 8 focuses on code execution on the modem. It discusses the specifics of the modem CPU's internal architecture and outlines the method for executing arbitrary code within the context of any modem OS process.

Section 9 contains a list of all the discovered vulnerabilities with their respective CVEs and criticality scores.

3 Modem description

The study focused on the EHS5-E series modem, originally manufactured by Thales before the business unit was acquired by Telit. Several modem models from this vendor share similar software and hardware architectures. Therefore, the findings of this study apply to devices across multiple model series:

- Cinterion BGS5
- Cinterion EHS5/6/7
- Cinterion PDS5/6/8
- Cinterion ELS61/81
- Cinterion PLS62

According to the software model, the modem consists of four software components:

- Firmware (FW)
- Application (App)
- Java Remote Control (JRC)
- Service LWM2M Agent (SLAE)

3.1 Modem software components

The modem is provided to device developers along with an SDK for creating software components that execute business logic, known as MIDlets. The firmware (FW) and application (App) components form part of the modem's low-level code, containing the modem's operating system and the execution environment for user MIDlets. A MIDlet is a Java application supported by a specialized subsystem, Java ME (Micro Edition), which features a limited set of Java commands. The JRC and SLAE components are special MIDlets developed by the manufacturer.

The user is provided with the ability to install MIDlets independently and to configure the security settings for their execution. The following security mechanisms are used for MIDlets:

- Java bytecode checks during installation (always enabled)
- MIDlet digital signatures (configured by the endpoint developer)

By default, only the manufacturer certificate is installed on the modem to validate MIDlets with manufacturer-level execution privileges. Installing and configuring certificates for custom MIDlets is the responsibility of the end product developer. This is described in more detail in the user manual[3].

3.2 Types of MIDlets

Based on our analysis, all MIDlets on the modem can be divided into two categories by privilege level:

- Manufacturer

---

[3] EHSx Java User's Guide, v15

- User signed / unsigned

Only the JRC and SLAE MIDlets belong to the manufacturer level. They have the highest privileges without any code execution restrictions at the Java level.

The second category of privileges is granted to user MIDlets. Their functionality is restricted in relation to file system (FS) operations, GSM module operations, etc. For example, a user MIDlet cannot read the entire modem FS, but the JRC module can.

If a user certificate is installed, only a signed user MIDlet with User Signed privileges will be executed on the modem. In other words, User Signed MIDlets are only used to protect the modem from executing a MIDlet from an illegitimate user, such as an intruder or security researcher.

3.3 Software update

A modem software update file is supplied by the manufacturer in the form an encrypted binary image. The image contains an update for the FW and App, an update for the JRC component, and optionally for the SLAE. Although the JRC is a MIDlet, it also contains modules that use APIs to interact with OS drivers. This provides interaction with OS subsystems: the FS, the data transfer subsystem, and communication with external hardware interfaces (e.g., USB). That's why the JRC and App components are always updated together – a driver update in the OS may entail a change in the APIs.

Our research did not involve a detailed analysis of the modem software update mechanism.

3.4 Installing MIDlets on the modem

MIDlets can be installed both locally and remotely. Local installation of MIDlets is done through the JRC component. Remote installation is possible via a special OTAP mechanism, or in M2M scenarios, via the SLAE component.

Using the modem in an M2M scenario involves creating a personal user account on the manufacturer's website. This personal account allows the user to perform standard actions with MIDlets on all paired devices, such as installing and uninstalling MIDlets. In our study, we did not analyze the mechanisms of remote M2M installation of MIDlets.

OTAP support is provided by the App and JRC components. The process of installing and updating user MIDlets via OTAP requires prior activation of OTAP by the user on the modem. The user may specify additional attributes that will be used for OTAP: *JAD File URL*, *HTTP User*, *HTTP Password*, etc. The process of updating via OTAP is described in detail in the EHSx Java User's Guide.

Local installation is performed via the MES communication protocol and special AT commands. The interface itself and AT command processing are implemented in the JRC component. The MES protocol enables interaction with the modem's user FS (hereinafter referred to as the "UFS"), including writing to or deleting files from the UFS.

Upon installing the driver provided with the SDK, the content of the modem's UFS, which is mounted at the path '///a:/', becomes accessible. The driver functions as a user add-on over the MES protocol. Although MES formally supports working with any path value except for '///a:/', no other internal paths are known, and attempts to read paths starting with a different root result in an error. This is due to the filtering of query parameters in MES, ensuring that

everything not belonging to the 'a://' root returns an error, even though the UFS has several valid roots.

Local installation of MIDlets is performed in two steps. First, the MIDlet files (*.jar* and *.jad*) must be copied to the modem UFS. Next, the MIDlet is installed on the modem using the *AT^SJAM=0* AT command. During the execution of this command, the MIDlet files are copied to a part of the modem FS that is inaccessible to the user and are then removed from the UFS. The path to which MIDlets are copied during installation is unknown. This helps ensure the confidentiality of both user and manufacturer MIDlets. An installed MIDlet is launched from its new location in the modem FS. A list of all installed MIDlets can also be extracted using the *AT^SJAM* command. Figure 1 shows an example of the output of this command.

```
^SJAM: "SLAE.jad","SL Agent Module Services","Gemalto M2M GmbH","2.2.0",0,493043,0,42
^SJAM: "a:/JRC-1.60.00.jad","Java Remote Control MIDlet Suite","Cinterion","1.60.00",1,630885,0,1
```

Figure 1. Example output of the AT^SJAM command

The modem documentation (Figure 2) states that the *javax.microedition.io.file.File.FileConnection* connector used to work with the FS filters requests to files with the *.jar* extension.

Cinterion WM

In BGSx EHSx PDSx series product, any attributes (readable, writeable, hidden) are not supported. In ELS61-x ELS81-x PLS52-x PLS62-x series product, attributes (writeable, hidden) are supported, readable is not supported. Still for compatibility reasons the corresponding set and get methods do not throw Exceptions when used. Files are always reported as being readable, writeable and not hidden.
Modification date is only supported for files. The method lastModified() will return 0 when used on a directory.
Files with extension ".jar" are protected. They can be opened only in WRITE mode, e.g.: `Connector.open(name, Connector.WRITE)`

Figure 2. Snippet from the Cinterion documentation

This behavior was confirmed by a simple test: trying to access files with the *.jar* extension (Figure 3) produced an error.

```
java.lang.SecurityException: Application not authorized to access the restricted API:javax.microedition.io.Connector.file.man
ufacturer
 - com.sun.midp.security.SecurityHandler.checkForPermission(), bci=147
 - com.sun.midp.security.SecurityHandler.checkForPermission(), bci=26
 - com.sun.midp.midletsuite.MIDletSuiteImpl.checkForPermission(), bci=20
 - com.sun.midp.midletsuite.MIDletSuiteImpl.checkForPermission(), bci=18
 - com.sun.midp.main.CldcAccessControlContext.checkPermissionImpl(), bci=34
 - com.sun.j2me.security.AccessControlContextAdapter.checkPermission(), bci=4
 - com.sun.j2me.security.AccessController.checkPermission(), bci=29
 - com.sun.j2me.app.AppPackage.checkForPermission(), bci=31
 - com.sun.io.j2me.file.Protocol.checkPermission(), bci=80
 - com.sun.io.j2me.file.Protocol.checkManufacturerPermission(), bci=42
 - com.sun.io.j2me.file.Protocol.openPrimImpl(), bci=603
 - com.sun.io.j2me.file.Protocol.openPrim(), bci=5
 - javax.microedition.io.Connector.open(), bci=47
 - javax.microedition.io.Connector.open(), bci=3
 - javax.microedition.io.Connector.open(), bci=2
 - WTKSamples.helloworld.HelloWorld.startApp(HelloWorld.java:101)
 - javax.microedition.midlet.MIDletTunnelImpl.callStartApp(), bci=1
 - com.sun.midp.midlet.MIDletPeer.startApp(), bci=5
 - com.sun.midp.midlet.MIDletStateHandler.startSuite(), bci=261
 - com.sun.midp.main.AbstractMIDletSuiteLoader.startSuite(), bci=38
 - com.sun.midp.main.CldcMIDletSuiteLoader.startSuite(), bci=5
 - com.sun.midp.main.AbstractMIDletSuiteLoader.runMIDletSuite(), bci=134
 - com.sun.midp.main.AppIsolateMIDletSuiteLoader.main(), bci=26
destroyApp(true)
MIDlet:WTKSamples.helloworld.HelloWorld abnormal exit
```

Figure 3. Attempting to access files with the .jar extension

3.5 Debugging MIDlets and modem execution

According to the modem's official documentation[4], several interfaces are available to interact with the modem, as shown in Figure 4.

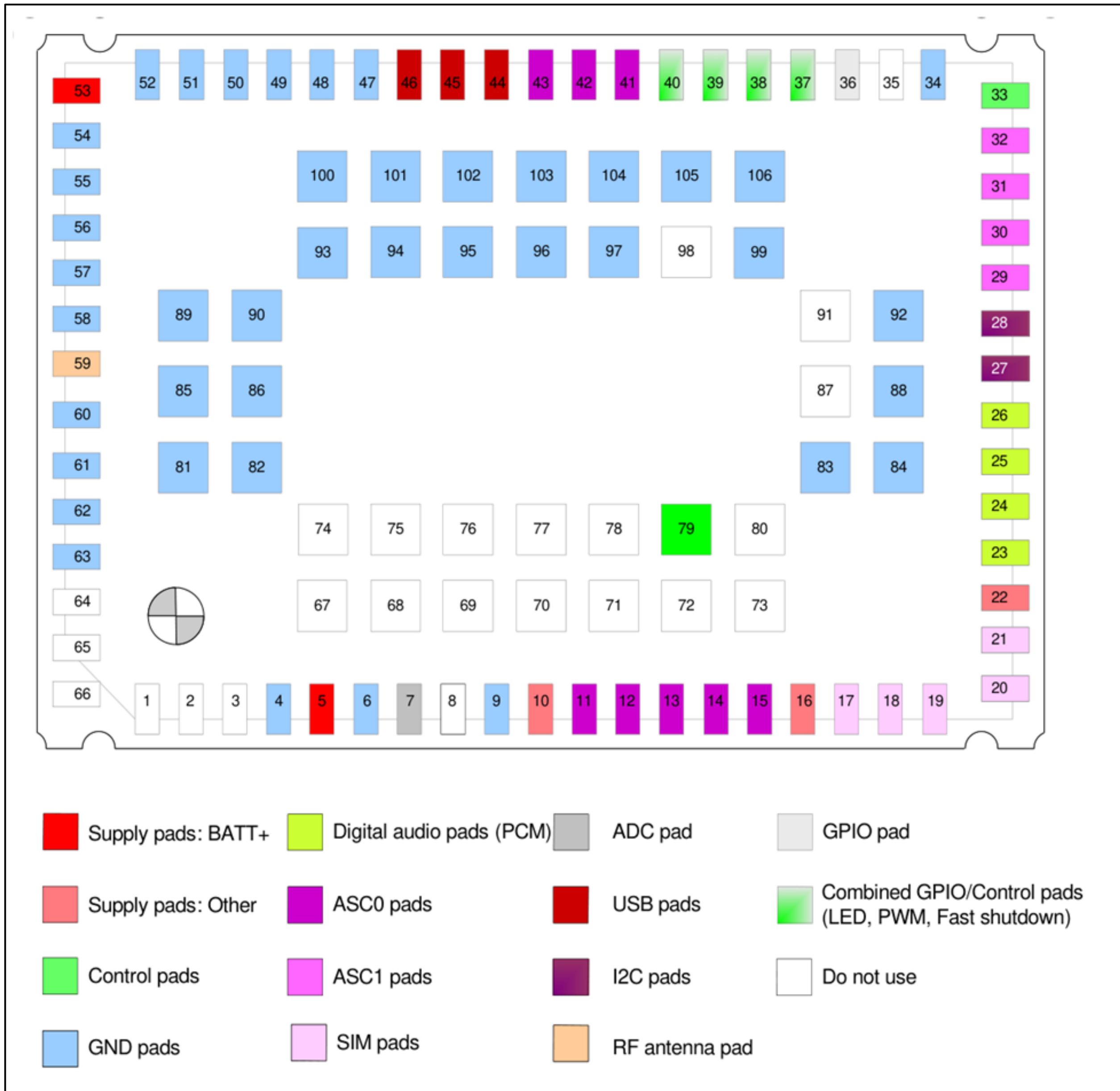


Figure 4. Available modem interfaces

Among these, the ASC0, ASC1 and USB interfaces make it possible to get debug information. ASC0 or ASC1 are used to transmit information between the modem and the debugging host through the UART protocol. In case of USB, the UART interface is emulated.

The modem provides multiple mechanisms for getting debug information:

[4] Cinterion® EHS5-E/EHS5-US Hardware Interface Description: https://www.gs-m2m.de/fileadmin/Bilder/GSM_Module/Module/EHS5/ehs5_us_e_hid_04003a.pdf

- IDE
- UART interface (logical or physical)

IDE allows full debugging of an executable MIDlet on the modem using the Java MIDlets debugging subsystem. This includes their loading onto the modem, step-by-step execution, and unloading. Interaction with the modem in debug mode takes place through a special PPP connection established via USB. The debugging mechanism is described in detail in the manufacturer's documentation[5].

In addition to debugging MIDlets, the modem allows the collection of logs of its subsystems, including MIDlet operations. The output format is determined by the *AT+TRACE* command. An example of the command to enable output of debugging information on the modem is shown below.

```
at+trace=,115200,"st=0,pr=1,bt=0,ap=1,db=1,lt=0,li=0"
```

The command parameters determine which modem subsystems to collect information from. Information on the subsystems from which debugging information can be collected is available in the detailed help (Figure 5) for this AT command.

[5] EHSx Java User's Guide, v15

```
at+trace=?
+TRACE: description START


at+trace=[<mode>],[<speed>],["<unit>=<umode>[,<unit>=<umode>[;...]]]",["<method>"],[PowerSavingCountdown]

<mode>:
-------------------------------------------------------------
0:        sets all units OFF [param <unit> will be ignored !]
1:        sets all units ON  [param <unit> will be ignored !]
no param: 3rd param. <units> configures trace-units
          -> trace? will then display 128 as <mode>

<speed>: (115200,230400,460800,921600,1843200,3000000,3250000,6000000)


<units>:
-------------
ap: apoxi
st: stack
db: debug
pr: printf
bt: bluetooth
lt: LLT
li: LwIP
gt: GATE
ae: AENEAS

<umode>:
-----------------
0: unit-trace OFF
1: unit-trace ON


<method>:
--------------------------------
"BTM":  byte stuffing trace method
"DTM":  direct trace method
"EBTM": extended byte stuffing trace method
"EDTM":extended direct trace methon(16bits)
"GEDTM":extended direct trace methon(16bits) with global time stamp


<PowerSavingCountdown in msecs>: (0-30000)


i.e.:
-------------------------------------------------
at+trace=0
at+trace=,460800
at+trace=,115200,"st=1,pr=1,bt=1,ap=0,db=1,lt=0,li=0"
at+trace=,,"lt=1,db=1,ga=0"
at+trace=,,,"EBTM"
at+trace=,,,,2000

+TRACE: description END

OK
```

Figure 5. Detailed help output of the AT+TRACE command

4 Information security assumptions regarding the modem

The modem's information security can be divided into two domains:

- Security of user MIDlets and manufacturer MIDlets
- Security of the modem OS

4.1 MIDlet security

MIDlets are protected on the modem to preserve two main security properties: confidentiality and integrity.

MIDlet confidentiality is provided by special restrictions on AT commands that allow the user to install, delete and execute MIDlets, as well as by prohibiting reading of .jar files from the UFS. After installation, MIDlets themselves are stored in an unknown location in the modem FS.

The question of violating the confidentiality of user MIDlets can be examined by testing two assumptions. The first is that it is impossible to determine the path where MIDlets are stored within the modem. The second asserts that it is impossible to bypass the restrictions preventing the reading of files with the *.jar* extension. If both assumptions are false, then MIDlet confidentiality could be violated.

The integrity property is ensured on the modem by means of a digital signature. The integrity of the Java byte code is additionally checked by the Java virtual machine for .jar files. However, this latter mechanism does not provide sufficient tamper protection, so we only consider the digital signature mechanism as a measure that ensures integrity.

Digital signatures are checked when MIDlets are launched. MIDlets with specific permissions can only be launched if this check is passed successfully. It is important to note that if the developer of the end device has not installed a security certificate, integrity is only ensured for the manufacturer's MIDlets that have been signed with its key and launched with the manufacturer's privileges. If a security certificate is installed, such protection is also extended to user MIDlets.

Thus, claiming that the integrity of user MIDlets cannot be violated entails there is no way to modify an existing MIDlet on the modem FS and that there is no way to launch a modified MIDlet if a digital signature is used (since in case that no signature is used, any user can install, reinstall and launch MIDlets). If both statements are false, the integrity of modem MIDlets could be violated.

4.2 Modem OS security

From a security perspective, the main component of a modem is its OS. Confidentiality and integrity are ensured by distributing OS updates only to registered users and only in encrypted form. We have not been able to find any update images available for download in public sources.

It is important to note that violating the confidentiality property alone does not grant an attacker the ability to execute arbitrary code on the modem. Similarly, even with the ability

to violate the integrity of the executable code, exploiting this capability to execute arbitrary code is significantly more difficult without access to the code base itself, either in source code or binary form. However, compromising both confidentiality and integrity would completely compromise the modem OS and all its MIDlets.

kaspersky

5 Obtaining modem firmware

While researching the modem, our team aimed to verify its stated security properties. We focused on analyzing both MIDlet and modem OS security. To achieve this, we developed a research device using a printed circuit board of our own design (Figure 6).

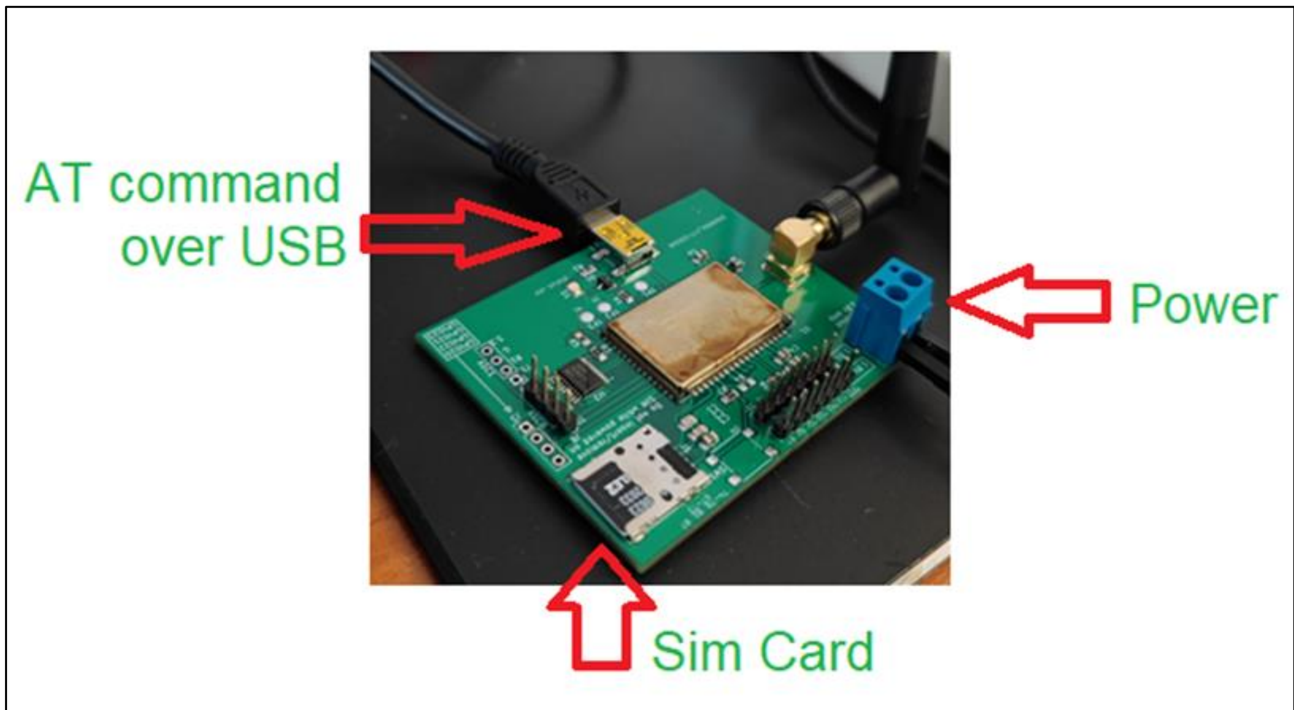


Figure 6. Research device

Next, we analyzed the modem’s hardware components (Figure 7). We identified its ROM, which is based on NAND memory and used to store the modem's software components. We extracted an image from this NAND memory using the ChipProg programmer.

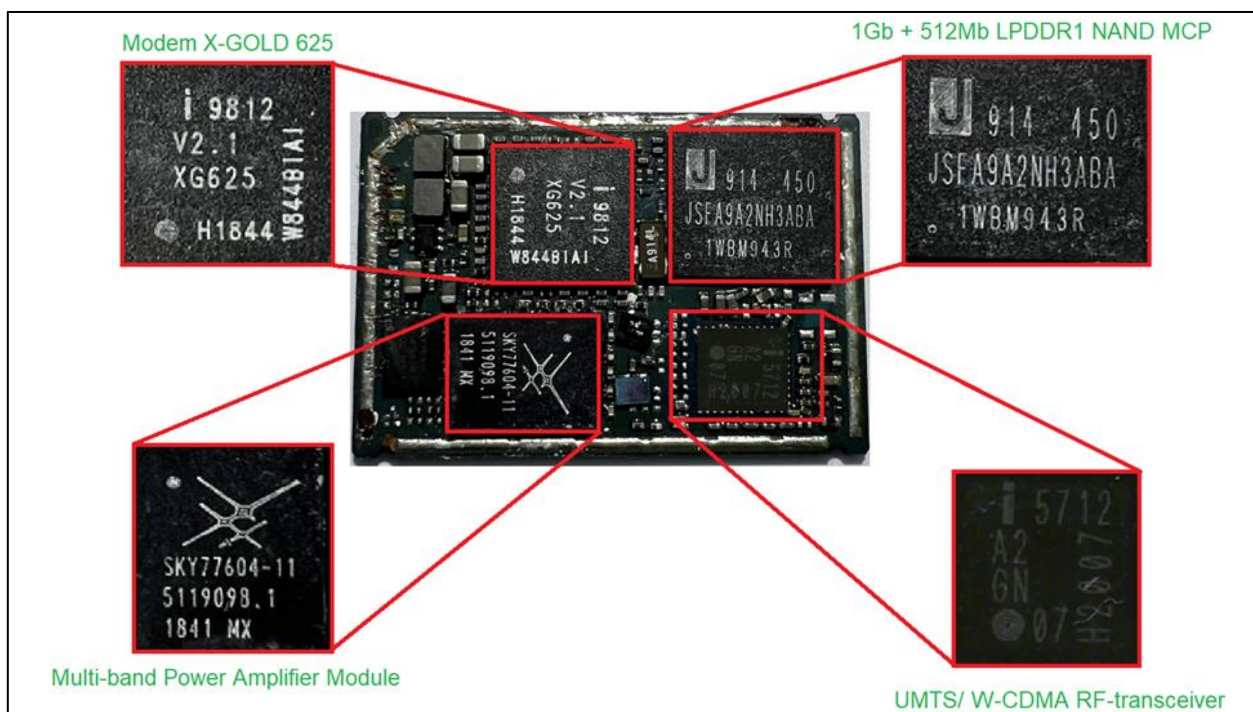


Figure 7. Hardware components of the modem

5.1 Building a logical image of firmware

The modem firmware extracted from the NAND memory dump consists of "raw" data from the physical NAND blocks. Data in NAND memory is stored as charges in specialized hardware cells, each made up of a transistor and capacitor. All physical space available for storage is segmented into sectors, which are then grouped into pages. Pages in turn are assembled into blocks[6].

Due to the nature of the storage device, reading errors may occur at the physical level - a zero may be read as a one and vice versa. Error-correcting codes are used to handle such errors. Usually, these codes are stored together with user data in NAND memory. The codes themselves are calculated for each sector of stored data and are stored together with other technical information in memory areas called the Spare Area (SA). As each sector is read, it is "corrected" with data from its SA, if necessary.

In addition, NAND memory itself is not durable. The high number of possible writes is achieved by two other data conversion mechanisms: gamma and wear leveling. Gamma is the bitwise XOR of a special gamma function with the data currently being written. The gamma is generated using a conventional LFSR[7] and is not an element of cryptographic protection but is used to ensure even wear of all memory storage cells. The wear leveling mechanism is used to ensure the uniform wear not only of the memory cells within sectors, but also of the physical sectors. To achieve this, the concepts of logical sector, page and block are introduced. A logical sector, page or block is mapped to actual physical blocks, pages and sectors. In the case of sector mapping, we speak of mapping a physical sector

---

[6] Open NAND Flash Interface Specification URL: https://media-www.micron.com/-/media/client/onfi/specs/onfi_4_2-gold.pdf

[7] Cryptographic Boolean Functions and Applications URL: https://doi.org/10.1016/C2016-0-00852-5

with some number, which we will denote as PS_N, into a logical sector with the number LS_N. In the case of blocks, a physical block with the number PB_N is completely mapped to a logical block with the number LB_N. Similarly, at the page level, a physical page PP_N is mapped to a logical LP_N.

Typical tasks for restoring user data from "raw" bytes include removal of gamma (if used), and recovering the physical-to-logical mapping. This is usually done by the NAND controller, but in this case, we had no access to it. Analysis of the entropy of the stripped image (Figure 8) indicated the absence of gamma.

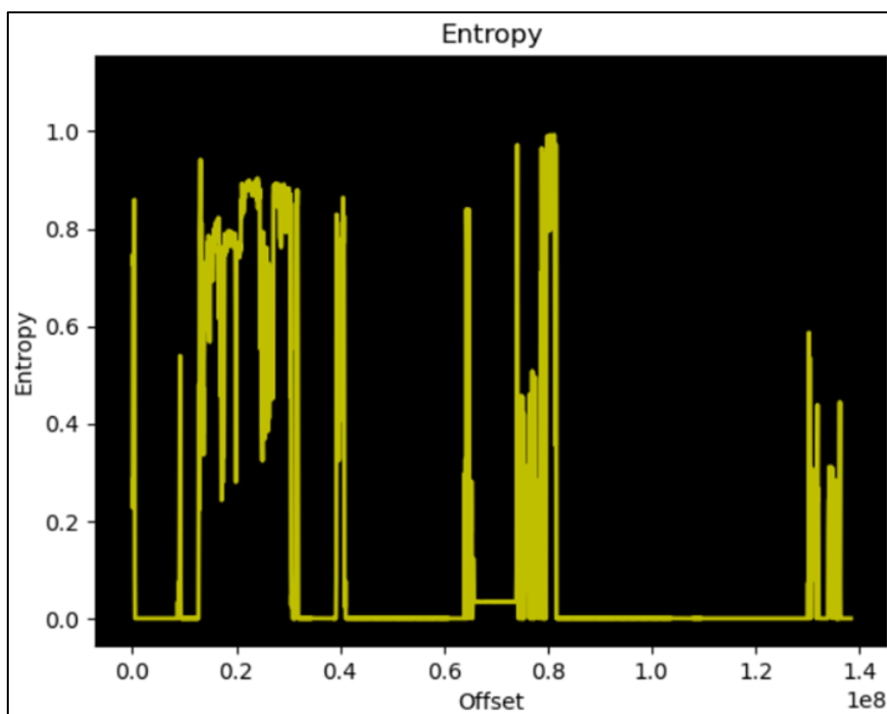


Figure 8. Entropy of the NAND controller image

However, the image itself did not look like meaningful binary data. It lacked artifacts of any physical storage-level file systems. *binwalk* also found only high-level file system artifacts. Accordingly, we concluded we were dealing with a standard mapping from the logical level of data representation to the physical level.

The mapping algorithm is not universal and depends on the vendor. There are solutions for recovering user data from "raw" data. Lacking such equipment, we decided to perform the data recovery on our own.

First, we determined where exactly the SA was stored. In our case, it turned out that every 0x200 bytes of data was accompanied by 16 bytes that clearly had nothing to do with user data. This can be seen particularly well at the boundary of the two physical sectors (Figure 9).

```
0000021A10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A40: 00 00 00 00 FF FF 88 10  00 10 41 00 04 00 28 28      ÿÿˆ► ►A ♦ ((
0000021A50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A80: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021A90: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AA0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AB0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AC0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AD0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AE0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021AF0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B80: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021B90: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BA0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BB0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BC0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BD0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BE0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021BF0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C50: 00 00 00 00 FF FF 88 10  00 10 41 00 05 00 48 B7      ÿÿˆ► ►A ♣ H·
0000021C60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000021C70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Figure 9. The boundary of the two physical sectors in the image

To analyze the contents of these 16 bytes, we wrote a script that built a binary image containing only this data for each physical sector of the original image. We assumed that these 16 bytes represented some structure containing service information. By reviewing the contents of the resulting file, we were able to partially determine the purpose of the fields of this structure. It turned out that in most cases the mapping included only “physical blocks to logical blocks” and “physical sectors to logical sectors”. In this case, the SA of a particular physical sector stores the logical block number LB_N and logical sector number LS_N to which this physical sector is mapped. The description is presented in Figure 10.

```
                                  LBN         LSN
A2 08 A2 08 FF FF DD 01 | 00 01 A0 00 FA 0B 22 22
58 0F A7 30 FF FF DD 01 | 00 01 A0 00 FB 0B 53 AC
26 06 D9 39 FF FF DD 01 | 00 01 A0 00 FC 0B 30 30
8E 03 8E 03 FF FF DD 01 | 00 01 A0 00 FD 0B 41 BE
BF 02 BF 02 FF FF DD 01 | 00 01 A0 00 FE 0B 40 BF
59 04 A6 3B FF FF DD 01 | 00 01 A0 00 FF 0B 31 31
AE 01 51 3E FF FF DD 01 | 00 01 A0 00 00 0C 4A B5
1B 0A 1B 0A FF FF DD 01 | 00 01 A0 00 01 0C 2B 2B
CE 01 CE 01 FF FF DD 01 | 00 01 A0 00 02 0C 2A 2A
75 09 75 09 FF FF DD 01 | 00 01 A0 00 03 0C 5A A5
E7 05 18 3A FF FF DD 01 | 00 01 A0 00 04 0C 28 28
9E 08 61 37 FF FF DD 01 | 00 01 A0 00 05 0C 48 B7
CA 03 CA 03 FF FF DD 01 | 00 01 A0 00 06 0C 48 B7
96 09 96 09 FF FF DD 01 | 00 01 A0 00 07 0C 28 28
59 07 A6 38 FF FF DD 01 | 00 01 A0 00 08 0C 39 39
C7 0F 38 30 FF FF DD 01 | 00 01 A0 00 09 0C 59 A6
BA 0A 45 35 FF FF DD 01 | 00 01 A0 00 0A 0C 58 A7
79 0A 86 35 FF FF DD 01 | 00 01 A0 00 0B 0C 38 38
```

Figure 10. Logical block LB_N and logical sector LS_N

Then, we wrote a script that builds binary images of logical blocks using data from service records. This did not assemble the full logical image, because some sectors appeared to be distributed into logical pages (PB_Ns). Assembling them required a deeper analysis of the physical block, page and sector allocation table structure. As a result, it turned out that few logical blocks were originally written on the disk. A separate binary file was generated for each of them, shown in Figure 11.

```
cinterion_block_0.bin
cinterion_block_1.bin
cinterion_block_2.bin
cinterion_block_9.bin
cinterion_block_11.bin
cinterion_block_14.bin
cinterion_block_16.bin
cinterion_block_17.bin
cinterion_block_22.bin
cinterion_block_23.bin
cinterion_block_24.bin
cinterion_block_61459.bin
```

Figure 11. Binary files for logical blocks

Not all blocks contained meaningful information, but we were able to identify blocks corresponding to the two main components of the modem: a combination of the FW and APP, and the UFS. The FW and APP components were in block number 1. We determined this by looking at the strings found in the assembled image (Figure 12). Furthermore, we observed that the resulting assembly was for the XMM6260 series modem.

```
0000041800: 00 00 00 00 21 53 59 53 | 54 45 4D 5F 56 45 52 53      !SYSTEM_VERS
0000041810: 49 4F 4E 20 44 41 54 41 | 20 53 54 52 55 43 54 55  ION DATA STRUCTU
0000041820: 52 45 21 00 00 00 00 00 | FA 00 00 00 66 11 BC 62  RE!      ú   f◄¼b
0000041830: 74 10 BC 62 D3 10 BC 62 | F5 10 BC 62 F5 10 BC 62  t►¼bÓ►¼bõ►¼bõ►¼b
0000041840: 0A 11 BC 62 0B 11 BC 62 | 0B 11 BC 62 0B 11 BC 62  ◙◄¼b♂◄¼b♂◄¼b♂◄¼b
0000041850: 0B 11 BC 62 0B 11 BC 62 | 0B 11 BC 62 0B 11 BC 62  ♂◄¼b♂◄¼b♂◄¼b♂◄¼b
0000041860: 0B 11 BC 62 0B 11 BC 62 | 7B 00 00 00 60 12 BC 62  ♂◄¼b♂◄¼b{   `↕¼b
0000041870: 16 11 BC 62 20 20 20 20 | 58 4D 4D 36 32 36 30 5F  ▬◄¼b    XMM6260_
0000041880: 56 32 5F 4C 41 52 47 45 | 42 4C 4F 43 4B 5F 4E 41  V2_LARGEBLOCK_NA
0000041890: 4E 44 5F 44 41 54 41 43 | 41 52 44 5F 52 45 56 5F  ND_DATACARD_REV_
00000418A0: 32 2E 31 30 20 32 30 32 | 30 2D 4D 61 72 2D 32 34  2.10 2020-Mar-24
00000418B0: 20 31 38 3A 32 34 3A 30 | 39 20 0A 20 20 20 20 50   18:24:09 ◙    P
00000418C0: 44 42 5F 4E 4F 54 5F 41 | 56 41 49 4C 41 42 4C 45  DB_NOT_AVAILABLE
00000418D0: 20 0A 00 4D 4F 44 5F 36 | 32 36 30 5F 56 30 35 2E   ◙ MOD_6260_V05.
00000418E0: 31 34 31 37 2E 30 30 5F | 52 30 38 2E 31 5F 56 43  1417.00_R08.1_VC
00000418F0: 54 43 58 4F 00 20 20 20 | 20 20 20 20 20 20 20 20  TCXO
0000041900: 20 20 20 20 20 20 20 20 | 20 00 00 3C 6E 6F 20 6C             <no l
0000041910: 61 62 65 6C 3E 00 61 33 | 66 62 34 33 39 36 00 00  abel> a3fb4396
```

Figure 12. Strings found in the image

The correctness of the constructed image was confirmed by identifying various strings related to the modem OS libraries (Figure 13). These strings are located consecutively across several sectors, and after our transformations, we got a meaningful set of library names.

```
0000041E40: 63 68 65 63 6B 73 75 6D | 5F 74 6F 74 61 6C 00 61  checksum_total a
0000041E50: 2D 67 70 73 2E 6C 69 62 | 00 61 6C 69 2E 6C 69 62  -gps.lib ali.lib
0000041E60: 00 61 73 6E 31 2E 6C 69 | 62 00 62 69 70 68 2E 6C   asn1.lib biph.l
0000041E70: 69 62 00 63 61 74 5F 73 | 79 73 74 65 6D 2E 6C 69  ib cat_system.li
0000041E80: 62 00 63 64 64 2E 6C 69 | 62 00 63 65 75 2E 6C 69  b cdd.lib ceu.li
0000041E90: 62 00 63 6D 2E 6C 69 62 | 00 63 6F 6E 6E 6D 6E 67  b cm.lib connmng
0000041EA0: 72 2E 6C 69 62 00 63 70 | 6C 61 6E 65 2E 6C 69 62  r.lib cplane.lib
0000041EB0: 00 63 70 73 5F 61 70 69 | 2E 6C 69 62 00 63 73 69   cps_api.lib csi
0000041EC0: 5F 67 74 6F 36 32 36 30 | 76 35 2E 6C 69 62 00 63  _gto6260v5.lib c
0000041ED0: 73 6E 31 2E 6C 69 62 00 | 64 68 63 70 76 36 2E 6C  sn1.lib dhcpv6.l
0000041EE0: 69 62 00 64 69 73 70 61 | 74 63 68 65 72 2E 6C 69  ib dispatcher.li
0000041EF0: 62 00 64 72 2E 6C 69 62 | 00 64 72 69 76 65 72 2E  b dr.lib driver.
0000041F00: 6C 69 62 00 64 72 76 5F | 32 67 5F 70 73 5F 64 72  lib drv_2g_ps_dr
0000041F10: 69 76 65 72 73 2E 6C 69 | 62 00 64 72 76 5F 33 67  ivers.lib drv_3g
0000041F20: 5F 70 73 5F 64 72 69 76 | 65 72 73 2E 6C 69 62 00  _ps_drivers.lib
0000041F30: 64 72 76 5F 61 67 70 73 | 2E 6C 69 62 00 64 72 76  drv_agps.lib drv
0000041F40: 5F 61 75 64 69 6F 5F 6D | 6D 2E 6C 69 62 00 64 72  _audio_mm.lib dr
0000041F50: 76 5F 61 75 64 69 6F 5F | 6D 6F 64 65 6D 2E 6C 69  v_audio_modem.li
0000041F60: 62 00 64 72 76 5F 62 61 | 74 74 65 72 79 5F 6D 61  b drv_battery_ma
0000041F70: 6E 61 67 65 6D 65 6E 74 | 2E 6C 69 62 00 64 72 76  nagement.lib drv
0000041F80: 5F 63 6F 6E 6E 65 63 74 | 69 76 69 74 79 2E 6C 69  _connectivity.li
0000041F90: 62 00 64 72 76 5F 63 6F | 72 65 5F 64 72 69 76 65  b drv_core_drive
0000041FA0: 72 73 2E 6C 69 62 00 64 | 72 76 5F 64 72 69 76 65  rs.lib drv_drive
0000041FB0: 72 5F 6C 69 62 72 61 72 | 69 65 73 2E 6C 69 62 00  r_libraries.lib
0000041FC0: 64 72 76 5F 6D 65 6D 6F | 72 79 5F 6D 61 6E 61 67  drv_memory_manag
0000041FD0: 65 6D 65 6E 74 2E 6C 69 | 62 00 64 72 76 5F 6E 76  ement.lib drv_nv
0000041FE0: 5F 6D 65 6D 6F 72 79 5F | 64 72 69 76 65 72 2E 6C  _memory_driver.l
0000041FF0: 69 62 00 64 72 76 5F 6F | 70 65 72 61 74 69 6F 6E  ib drv_operation
0000042000: 5F 6D 61 69 6E 74 65 6E | 61 6E 63 65 2E 6C 69 62  _maintenance.lib
0000042010: 00 64 72 76 5F 70 6F 77 | 65 72 5F 63 6F 6E 74 72   drv_power_contr
0000042020: 6F 6C 2E 6C 69 62 00 64 | 72 76 5F 70 6F 77 65 72  ol.lib drv_power
0000042030: 5F 6D 61 6E 61 67 65 6D | 65 6E 74 2E 6C 69 62 00  _management.lib
0000042040: 64 72 76 5F 72 74 63 2E | 6C 69 62 00 64 72 76 5F  drv_rtc.lib drv_
0000042050: 73 65 63 75 72 69 74 79 | 2E 6C 69 62 00 64 72 76  security.lib drv
0000042060: 5F 73 65 63 75 72 69 74 | 79 5F 64 73 6C 62 2E 6C  _security_dslb.l
```

Figure 13. Modem OS library name strings

The UFS image was easily identified by the characteristic file system header shown in Figure 14.

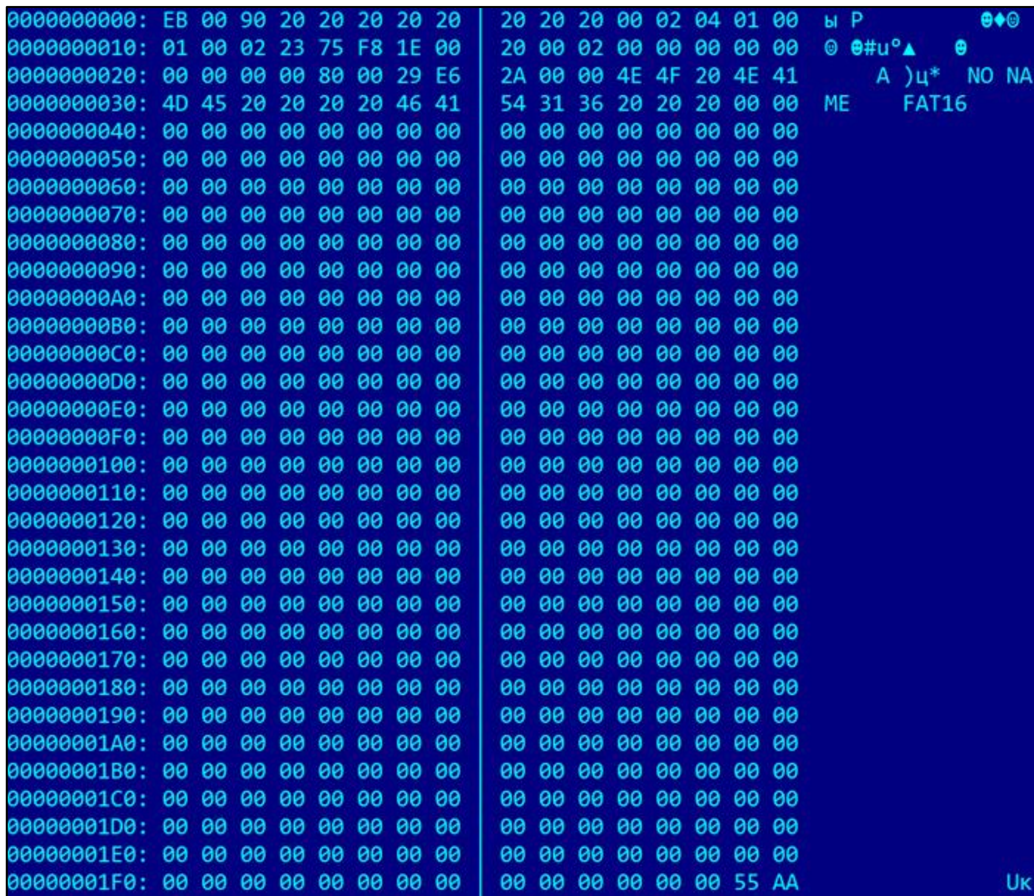


Figure 14. FS header

As a result, when we mounted the resulting image as a FAT FS image, we had access to all user MIDlets as well as system files and folders (Figure 15).

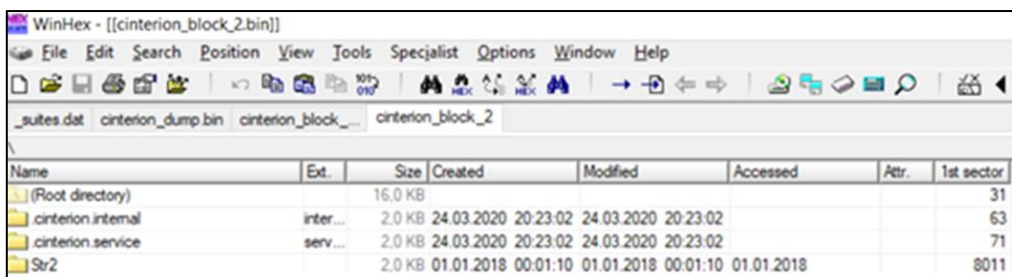


Figure 15. FAT FS

5.2 Analyzing the contents of the UFS

After reviewing the contents of the UFS, we found that it contained folders and files hidden from the user and inaccessible through the MES mechanism, namely *.cinterion.internal* and *.cinterion.service*. Access to them was restricted directly in the JRC MIDlet code, filtered by their name prefix. The code snippet from the JRC module that filters access to these folders is shown in Figure 16.

```
while(enumeration0.hasMoreElements()) {
    String s2 = (String)enumeration0.nextElement();
    if(s2.toLowerCase().indexOf(".cinterion.") >= 0) {
        continue;
    }
```

Figure 16. Example code from the JRC module

We discovered that the virtual root *///a:/* corresponds to the physical path */sys/*. We also extracted information about the path to which user MIDlets are copied after installation and the format in which they are stored. The contents of the folder with MIDlets is shown in Figure 17.

Name	Date	Type	Size
amsbackup	11/7/2022 3:37 PM	File folder	
backup	11/7/2022 3:37 PM	File folder	
_main.ks	11/4/2022 3:54 PM	KS File	1 KB
_suites.dat	11/4/2022 3:54 PM	DAT File	2 KB
_trans.dat	11/4/2022 3:54 PM	DAT File	1 KB
00000003.ap	11/4/2022 3:54 PM	AP File	5 KB
00000003.ii	11/4/2022 3:54 PM	II File	1 KB
00000003	11/4/2022 3:54 PM	Executable Jar File	612 KB
00000003.ss	11/4/2022 3:54 PM	SS File	1 KB
00000005.ap	11/4/2022 3:54 PM	AP File	5 KB
00000005.ii	11/4/2022 3:54 PM	II File	1 KB
00000005	11/4/2022 3:54 PM	Executable Jar File	477 KB
00000005.ss	11/4/2022 3:54 PM	SS File	1 KB
00000007.ap	11/4/2022 3:54 PM	AP File	2 KB
00000007.ii	11/4/2022 3:54 PM	II File	1 KB
00000007	11/4/2022 3:54 PM	Executable Jar File	287 KB
00000007.ss	11/4/2022 3:54 PM	SS File	2 KB
Otap_AtParams.bin	11/4/2022 3:54 PM	BIN File	2 KB

Figure 17. Contents of the folder with MIDlets

It was determined that all MIDlets are stored on the device at the path */sys/.cinterion.internal/java*. Each MIDlet is represented on the FS as four files with the *.ss*, *.ii*, *.ap*, and *.jar* extensions.

Files with the *.jar* extension are Java executable files. After analyzing the contents of the folder with installed MIDlets, we found that each MIDlet is renamed during the installation process. To allow the user to run MIDlets by their name, the file system maintains a mapping between the original name of the MIDlet and its alias within a simple database that is stored in the *_suites.dat* file. For example, by analyzing the binary contents of this file, it is clear that the file named *00000003.jar* (Figure 18) is actually *JRC.jar*.

```
30 00 00 00 2A 00 00 00 | 2F 00 73 00 79 00 73 00   0   *   / s y s
2F 00 2E 00 63 00 69 00 | 6E 00 74 00 65 00 72 00   / . c i n t e r
69 00 6F 00 6E 00 2E 00 | 69 00 6E 00 74 00 65 00   i o n . i n t e
72 00 6E 00 61 00 6C 00 | 2F 00 6A 00 61 00 76 00   r n a l / j a v
61 00 2F 00 30 00 30 00 | 30 00 30 00 30 00 30 00   a / 0 0 0 0 0 0
30 00 33 00 2E 00 6A 00 | 61 00 72 00 FF FF FF FF   0 3 . j a r ????
FF FF FF FF 03 00 00 00 | 00 00 00 00 00 00 00 00   ????♥
05 00 00 00 00 00 00 00 | 01 00 00 00 00 00 00 00   ♣       ☺
00 00 00 00 02 8D 09 00 | 65 A0 09 00 FF FF FF FF       ☻?○ e ○ ????
00 00 00 00 00 00 00 00 | 00 00 00 00 1C 00 00 00               ∟
63 00 6F 00 6D 00 2E 00 | 63 00 69 00 6E 00 74 00   c o m . c i n t
65 00 72 00 69 00 6F 00 | 6E 00 2E 00 6A 00 72 00   e r i o n . j r
63 00 2E 00 4A 00 52 00 | 43 00 5F 00 4D 00 69 00   c . J R C _ M i
64 00 6C 00 65 00 74 00 | 20 00 00 00 4A 00 61 00   d l e t     J a
76 00 61 00 20 00 52 00 | 65 00 6D 00 6F 00 74 00   v a   R e m o t
65 00 20 00 43 00 6F 00 | 6E 00 74 00 72 00 6F 00   e   C o n t r o
6C 00 20 00 4D 00 49 00 | 44 00 6C 00 65 00 74 00   l   M I D l e t
20 00 53 00 75 00 69 00 | 74 00 65 00 FF FF FF FF     S u i t e ????
09 00 00 00 43 00 69 00 | 6E 00 74 00 65 00 72 00   ○   C i n t e r
69 00 6F 00 6E 00 00 00 | 20 00 00 00 4A 00 61 00   i o n       J a
76 00 61 00 20 00 52 00 | 65 00 6D 00 6F 00 74 00   v a   R e m o t
65 00 20 00 43 00 6F 00 | 6E 00 74 00 72 00 6F 00   e   C o n t r o
6C 00 20 00 4D 00 49 00 | 44 00 6C 00 65 00 74 00   l   M I D l e t
20 00 53 00 75 00 69 00 | 74 00 65 00 07 00 00 00     S u i t e •
31 00 2E 00 36 00 30 00 | 2E 00 30 00 30 00 00 00   1 . 6 0 . 0 0
2A 00 00 00 20 00 00 00 | 4D 00 49 00 44 00 6C 00   *       M I D l
65 00 74 00 2D 00 56 00 | 65 00 72 00 73 00 69 00   e t - V e r s i
6F 00 6E 00 00 00 20 00 | FE CA 00 00 0C 00 00 00   o n     ??  ♀
32 00 2E 00 32 00 2E 00 | 30 00 00 00 FE CA 00 00   2 . 2 . 0   ??
```

Figure 18. Contents of the binary file _suites.dat

The *.ap* file is the *.jad* file converted to UTF-16. This file contains information about the original name of the MIDlet and the libraries that were used. It may also contain the MIDlet's digital signature. The contents of such a file for the JRC MIDlet are shown in Figure 19.

```
1  Manifest-Version: 1.0
2  MIDlet-Vendor: Cinterion
3  MIDlet-Version: 1.60.00
4  Oracle-MIDlet-Restart: false
5  Midlet-CertStore: firmware
6  Oracle-MIDlet-Autostart: 1
7  MicroEdition-Configuration: CLDC-1.1
8  MIDlet-1: JRC_Midlet,,com.cinterion.jrc.JRC_Midlet
9  Created-By: 1.7.0_07 (Oracle Corporation)
10 MIDlet-Name: Java Remote Control MIDlet Suite
11 MicroEdition-Profile: IMP-NG
```

Figure 19. Contents of the .ap file for the JRC MIDlet

The *.ii* file contains information about the MIDlet installation path, the permissions assigned during installation, and other service information (Figure 20).

```
1C 00 00 00 66 00 69 00 | 6C 00 65 00 3A 00 2F 00   └   f i l e : /
2F 00 2F 00 2F 00 73 00 | 79 00 73 00 2F 00 4A 00   / / / s y s / J
52 00 43 00 2D 00 31 00 | 2E 00 36 00 30 00 2E 00   R C - 1 . 6 0 .
30 00 30 00 2E 00 6A 00 | 61 00 64 00 1C 00 00 00   0 0 . j a d └
66 00 69 00 6C 00 65 00 | 3A 00 2F 00 2F 00 2F 00   f i l e : / / /
2F 00 73 00 79 00 73 00 | 2F 00 4A 00 52 00 43 00   / s y s / J R C
2D 00 31 00 2E 00 36 00 | 30 00 2E 00 30 00 30 00   - 1 . 6 0 . 0 0
2E 00 6A 00 61 00 72 00 | 0C 00 00 00 6D 00 61 00   . j a r ♀   m a
6E 00 75 00 66 00 61 00 | 63 00 74 00 75 00 72 00   n u f a c t u r
65 00 72 00 01 01 00 00 | 00 38 00 00 00 43 00 3D   e r ☺☺   8   C =
00 44 00 45 00 3B 00 53 | 00 54 00 3D 00 42 00 65    D E ; S T = B e
00 72 00 6C 00 69 00 6E | 00 3B 00 4C 00 3D 00 42    r l i n ; L = B
00 65 00 72 00 6C 00 69 | 00 6E 00 3B 00 4F 00 3D    e r l i n ; O =
00 43 00 49 00 4E 00 54 | 00 45 00 52 00 49 00 4F    C I N T E R I O
00 4E 00 3B 00 4F 00 55 | 00 3D 00 43 00 49 00 4E    N ; O U = C I N
00 54 00 45 00 52 00 49 | 00 4F 00 4E 00 3B 00 43    T E R I O N ; C
00 4E 00 3D 00 65 00 68 | 00 73 00 35 00             N = e h s 5
```

Figure 20. Contents of a file with the .ii extension

Finally, the *.ss* file contains a description of the Java-level permissions available to this MIDlet. An example of the JRC vendor MIDlet permissions description is shown in Figure 21.

```
00 00 00 00 00 01 00 00 | 00 1C 00 00 00 63 6F 6D        ☺   └   com
2E 73 75 6E 2E 6D 69 64 | 70 2E 4D 49 44 50 50 65   .sun.midp.MIDPPe
72 6D 69 73 73 69 6F 6E | 00 00                     rmission
```

Figure 21. Example of JRC vendor MIDlet permissions description

It is important to note that this permission set gives unrestricted access to the Java virtual machine's system classes and represents the manufacturer privilege level. Only two MIDlets have such privileges: JRC and SLAE. Any user MIDlet must list in its manifest the classes and methods that it needs access to at runtime. Part of the manifest for our test MIDlet is shown in Figure 22.

```
08 00 00 00 00 16 00 00 | 00 2A 00 00 00 6A 61 76   ◘   ▬   *   jav
61 78 2E 6D 69 63 72 6F | 65 64 69 74 69 6F 6E 2E   ax.microedition.
63 6F 6E 74 65 6E 74 2E | 43 6F 6E 74 65 6E 74 48   content.ContentH
61 6E 64 6C 65 72 00 01 | 24 00 00 00 6A 61 76 61   andler ☺$   java
78 2E 6D 69 63 72 6F 65 | 64 69 74 69 6F 6E 2E 69   x.microedition.i
6F 2E 43 6F 6E 6E 65 63 | 74 6F 72 2E 73 73 6C 00   o.Connector.ssl
01 29 00 00 00 6A 61 76 | 61 78 2E 6D 69 63 72 6F   ☺)   javax.micro
65 64 69 74 69 6F 6E 2E | 6D 69 64 6C 65 74 2E 4D   edition.midlet.M
49 44 6C 65 74 2E 69 6E | 73 74 61 6C 6C 00 01 31   IDlet.install ☺1
00 00 00 6A 61 76 61 78 | 2E 6D 69 63 72 6F 65 64      javax.microed
69 74 69 6F 6E 2E 69 6F | 2E 43 6F 6E 6E 65 63 74   ition.io.Connect
6F 72 2E 64 61 74 61 67 | 72 61 6D 72 65 63 65 69   or.datagramrecei
76 65 72 00 01 30 00 00 | 00 6A 61 76 61 78 2E 6D   ver ☺0   javax.m
69 63 72 6F 65 64 69 74 | 69 6F 6E 2E 6C 6F 63 61   icroedition.loca
74 69 6F 6E 2E 4C 61 6E | 64 6D 61 72 6B 53 74 6F   tion.LandmarkSto
72 65 2E 77 72 69 74 65 | 00 01 2A 00 00 00 6A 61   re.write ☺*   ja
76 61 78 2E 6D 69 63 72 | 6F 65 64 69 74 69 6F 6E   vax.microedition
2E 69 6F 2E 43 6F 6E 6E | 65 63 74 6F 72 2E 66 69   .io.Connector.fi
6C 65 2E 72 65 61 64 00 | 01 29 00 00 00 6A 61 76   le.read ☺)   jav
```

Figure 22. Example manifest for our test MIDlet

The *OTAP_AtParams.bin* file deserves special attention. Its name suggests that it may be related to OTAP configuration. During experiments, we found out that this file is created when the *AT^SJOTAP* command is executed and contains the settings specified in this command. Part of the file created during our tests is shown in Figure 23.

```
03 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00  ♥
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 | 00 68 74 74 70 3A 2F 2F          http://
31 37 32 2E 32 30 2E 31 | 30 2E 32 2F 74 65 73 74  172.20.10.2/test
6A 61 64 00 00 00 00 00 | 00 00 00 00 00 00 00 00  jad
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
```

Figure 23. Example of OTAP_AtParams.bin file contents

6 Analysis of MIDlets security

6.1 FTP client security

During our analysis of documented AT commands, we found that there is a set of AT commands dedicated to FTP interactions.

The FTP service allows downloading and uploading to any internal folder of the modem (Figure 24). JAR files can be accessed without restriction, because the FTP service is implemented in the code of the JRC MIDlet and operates in the manufacturer security domain. Thus, any system or user files, including MIDlets, can be read through this method.

10.15.14 FTP Download to FFS (URC Mode)

Configure the service profile 1 for FTP:

`AT^SISS=1,srvType,"Ftp"`	Select service type FTP.
`OK`	
`AT^SISS=1,conId,"0"`	Select connection profile 0.
`OK`	
`AT^SISS=1,address,"ftp://ftp.heise.de/pub"`	Specify FTP address.
`OK`	
`AT^SISS=1,cmd,"fget"`	Select command type download.
`OK`	
`AT^SISS=1,user,"anonymous"`	
`OK`	
`AT^SISS=1,passwd,"tester@google.com"`	
`OK`	
`AT^SISS=1,path,"file:///a:/data/"`	Specify target path on local FFS.
`OK`	
`AT^SISS=1,files,"INDEX"`	Specify file to be downloaded.
`OK`	

Figure 24. FTP operations

The API of the aforementioned FTP client is not protected and is publicly described in the modem's AT command documentation[8]. This means an attacker with physical access to the device can gain read / write access to any files and directories in the modem's file system, including the hidden ones.

6.2 Getting information about internal FS paths via a static method inside a custom MIDlet

A user MIDlet can call the static methods *Configuration.getSystemPropertiesNames()* and *Configuration.getSystemProperty(VAR_NAME)*. These methods return a list of all environment variables and the value of a particular environment variable, respectively.

JVM environment variables, such as the internal path to where MIDlets are stored, can be extracted this way (Figure 25).

[8] Cinterion® EHS5-E AT Command Set: https://teleofis.ru/upload/iblock/c85/c851073722a1d525ca5d49cf84e1404a.pdf

```
audio.samplerates
audio3d.simultaneouslocations
camera.orientations
camera.resolutions
supports.mediacapabilities
tuner.modulations
system.storage_root
profiler.filename
system.default_storage
com.oracle.midp.ams.headless.autostart.delaytime
com.oracle.midp.ams.headless.autostart.enable
com.oracle.highlevelui.theme.file
com.oracle.highlevelui.theme.name
com.oracle.jwc.version
javax.microedition.io.Connector.protocolpath
javax.microedition.xmlapi.events.version
javax.microedition.xmlapi.version microedition.configuration
microedition.io.file.FileConnection.version
microedition.jtwi.version microedition.locale
microedition.location.version
microedition.msa.version
microedition.platform
microedition.profiles
path.separator
security.messagefile
security.policyfile
xml.jaxp.subset.version
```

Figure 25. JVM environment variables

Calling the *Configuration.getSystemProperty* function with the "*system.storage_root*" argument from a user MIDlet returns the native path of all installed MIDlets. For our research, this was */sys/.cinterion.internal/java*.

6.3 Getting information about internal FS paths from a user MIDlet via a standard *FileConnector* call

Any user MIDlet can bypass the access restriction to the FS above the virtual root of the virtual FS tree. Access to files on the virtual FS is performed by calling the

```
(FileConnection) Connector.open("file:///root:/PATH")
```

method, where root is some virtual root of the FS. In our case, using a *%2E%2E%2F* (URL-encoded "*../*") sequence at the start of a PATH, forms the path "*native_path/../PATH*" for the FS access handler, where *native_path* is the actual location where the virtual root is mounted.

This occurs due to a flaw in the way the open function processes the path sequence: it first checks for any *"../"* sequences and only then converts the escape sequence to ASCII, allowing access to any part of the native FS.

We also found out that virtual root *b:/* is a folder inside virtual root *a:/* (Figure 26 and Figure 27).

```
Path exists: file:///a:/

Name is:
        Path is: /a:/

Path not exists: file:///a:/../

Name is: ../
        Path is: /a:/

Path not exists: file:///a:/../../

Name is: ../
        Path is: /a:/../

Path not exists: file:///a:/../../../
```

Figure 26. FS tree

```
Path exists: file:///b:/
Name is: Path is: /b:/
Path exists: file:///b:/../
Name is: ../Path is: /b:/
Path not exists: file:///b:/../../
Name is: ../Path is: /b:/../
Path not exists: file:///b:/../../../
Name is: ../Path is: /b:/../../
Path not exists: file:///b:/../../../../
Name is: ../Path is: /b:/../../../
```

Figure 27. FS tree

6.4 Changing the security domain of a custom MIDlet

As mentioned earlier, all installed MIDlets are stored in the modem FS as a set of four files under the path */sys/.cinterion.internal/java*. When a MIDlet is started, its security domain is checked in the *.ii* file. Then, depending on the specified domain, access rights are assigned based on the *.ss* file. There is no verification of the digital signature when launching a MIDlet that has the manufacturer-level security domain.

Since any user MIDlet can use the aforementioned *FileConnection* Java class, the MIDlet's security permissions and security level can be escalated. An installed user MIDlet can replace its own *.ii* and *.ss* files, so that it starts executing in the manufacturer security domain (Figure 28 and Figure 29).

```
Step 0: Creating new midlet files directory
Step 0: Succcess
Step 1: Getting midlet files from secret folder
Step 1: Complete
Step 2: Creating new midlet permissons files
Step 2.1: Creating new midlet .ii file
Found file: 0000002B.ii
Step 2.2: Succcess
Step 2.2: Creating new midlet .ss file
Step 2.2: Succcess
Step 2: Complete
Step 3: Update midlet permissons files
Step 3: Complete
destroyApp(true)
MIDlet:PocMiD exited
```

Figure 28. Running our MIDlet for the first time

```
Step 0: Creating new midlet files directory
Step 0: Succcess
Step 1: Getting midlet files from secret folder
Step 1: Complete
Step 2: Creating new midlet permissons files
Step 2.1: Creating new midlet .ii file
Found file: 0000002B.ii
Midlet already in manufacturer mode!
Found directory: file:///a:/
.cinterion.internal/

.cinterion.service/
```

Figure 29. Running our MIDlet for the second time

7 Modem OS security

7.1 Selecting interfaces for research

While acquiring the modem firmware, we compromised the confidentiality of its OS data through invasive and destructive actions.

It was important for us to evaluate whether both the confidentiality and integrity of the OS image are preserved in the absence of invasive actions. To this end, we considered the available software interfaces for interacting with the modem. We identified two of the most widespread: the AT command interface and SMS messages.

The AT command interface is a well-known method for controlling the modem. To exchange AT commands with a modem, one simply needs to connect to one of its wired interfaces, such as USB. The confidentiality and/or integrity of this interface could be compromised if there are security flaws.

The SMS messaging interface is available on any modem and can be used by knowing the subscriber number of the target modem within the cellular operator's network. However, it is not always possible to send binary SMS messages due to operator restrictions. Using a fake base station allows such limitations to be bypassed.

7.2 Researching modem AT commands

We started our research of the modem’s OS security by analyzing the commands available at the AT interface level. We decided to rely on fuzzing as our main research approach. Accordingly, we needed to:

- Build a fuzzing corpus from a set of supported commands and their parameters
- Be able to identify run-time errors and their causes

The set of AT commands can be determined from the manufacturer's documentation[9]. This method yields a corpus composed of only publicly available user AT commands. However, we are interested in getting an exhaustive list of all AT commands available in our modem OS image.

By analyzing the modem’s OS image, we determined that the modem is based on RTOS ThreadX ARM11. While the main drivers and platform are written in C, some portions, such as the USB stack, are written in C++.

According to the documentation, the commands available to the user start with the "*AT*" prefix followed by a special character: "+", "^" or "*%*". The commands themselves have four execution modes: read, write, execute, and test (help output). Searching the binary image, we identified a data section (Figure 30) consisting of structures corresponding to the following representation of AT commands.

[9] Cinterion® EHS5-E AT Command Set: https://www.euromobile.ru/upload/iblock/ca2/ehs5-e_rn_v04000.pdf

```
at_functions_start AT_CMD_Descriptor <aCmer, aMobileTerminat, 1, 1, sub_62C3B4D0+1, \
                                        ; DATA XREF: sub_62EB2438+3C↑o
                                        ; ROM:off_62EB24A4↑o
                          sub_62C3B470+1, AT_CMER_testCMD+1, 0, 0, 0> ; "+CMER" ...
                 AT_CMD_Descriptor <aCgsms, aSmsSelectServi, 1, 1, sub_62CA354C+1, \ ; "+CGSMS" ...
                          at_cgsms_read_cmd+1, sub_62CA35D4+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgd_1, aSmsDeleteSmsAt, 1, 1, sub_62CA379C+1, 0, \ ; "+CMGD" ...
                          sub_62CA3864+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgf, aSmsMessageForm, 1, 1, sub_62CA38C8+1, \ ; "+CMGF" ...
                          sub_62CA3894+1, sub_62CA391C+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgl_5, aSmsListMessage, 1, 0, sub_62CA3B50+1, 0, \ ; "+CMGL" ...
                          sub_62CA3B38+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgr_4, aSmsReadMessage, 1, 1, sub_62CA3BCC+1, 0, \ ; "+CMGR" ...
                          sub_62BD8AA2+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgs, aSmsSendSmsMess, 1, 1, sub_62CA3BE8+1, 0, \ ; "+CMGS" ...
                          sub_62BD8AA2+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmgw, aSmsWriteMessag, 1, 0, sub_62CA3D30+1, 0, \ ; "+CMGW" ...
                          sub_62BD8AA2+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmms, aSmsMoreMessage, 1, 1, sub_62CA3EDC+1, \ ; "+CMMS" ...
                          sub_62CA3EAC+1, sub_62CA3F60+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCmss, aSmsSendMessage, 1, 1, sub_62CA3FB0+1, \ ; "+CMSS" ...
                          sub_62CA3F9C+1, sub_62CA408C+1, 0, 0, 0>
                 AT_CMD_Descriptor <aCnma, aSmsNewMessageA, 1, 0, sub_62CA40A8+1, 0, \ ; "+CNMA" ...
                          sub_62CA420C+1, 0, 0, 0>
```

Figure 30. Representation of AT commands in binary image

By analyzing the custom AT command descriptor's contents, we identified over 200 unique commands. Using the test function, we identified exactly which parameters and how many parameters each command expects. We found text descriptions of the commands, shown in Figure 31, and used them to determine their purpose.

```
aXlgnvram       DCB "+XLGNVRAM",0       ; DATA XREF: ROM:63005A98↑o
aGpsReadResetPo DCB "GPS: Read/Reset positioning information",0
                                        ; DATA XREF: ROM:63005A98↑o
aXlcsshutdown   DCB "+XLCSSHUTDOWN",0   ; DATA XREF: ROM:63005AC0↑o
aGpsShutdownGns DCB "GPS: Shutdown GNSS engine",0
                                        ; DATA XREF: ROM:63005AC0↑o
aXlcstest       DCB "%XLCSTEST",0       ; DATA XREF: ROM:63005AE8↑o
aGpsAutomaticLi DCB "GPS: Automatic link setup",0
                                        ; DATA XREF: ROM:63005AE8↑o
aXlcssuplver    DCB "+XLCSSUPLVER",0    ; DATA XREF: ROM:63005B10↑o
aGpsSetLcsSuplV DCB " GPS: Set LCS SUPL version ",0
                                        ; DATA XREF: ROM:63005B10↑o
aXlcslstr       DCB "+XLCSLSTR",0       ; DATA XREF: ROM:63005B38↑o
aGpsLocationSer DCB " GPS: Location service trigger request ",0
                                        ; DATA XREF: ROM:63005B38↑o
aXlcssuplappid  DCB "+XLCSSUPLAPPID",0  ; DATA XREF: ROM:63005B60↑o
aGpsSetModifyAp DCB "GPS: Set/Modify Appplication Id Parameters",0
                                        ; DATA XREF: ROM:63005B60↑o
aXlcsaetta      DCB "+XLCSAETTA",0      ; DATA XREF: ROM:63005B88↑o
aGpsTargetAreas DCB " GPS: Target Areas of AreaEvent Trigger Session ",0
                                        ; DATA XREF: ROM:63005B88↑o
aXlcstttplr     DCB "+XLCSTTTPLR",0     ; DATA XREF: ROM:63005BB0↑o
aGpsTransferToT DCB " GPS: Transfer To Third Party Location Request ",0
```

Figure 31. Text descriptions of AT commands

Among the available AT commands, we managed to find the "*AT+XLOG*" command, which allowed reading information about software and hardware errors. With the help of this command, we were able to establish the cause of errors and locate the failure point in the modem code. An example of the output of this command is shown in Figure 32.

```
AT+XLOG=0
AT+XLOG=0

+XGENDATA: "MOD_6260_V05.1417.00_R08.1_VCTCXO    XMM6260_V2_LARGEBLOCK_NAND_DATACARD_REV_2.10 2018-Aug-27 12:27:47

    PDB_NOT_AVAILABLE

"

+XLOG: Exception Number: 1

 Trap Class:      0xEEEE  (SW EXCEPTION)

Identification: 2120 (0x0848)

 Date: 2018:1:1

 Time: 0:4:47

 File: drv_2g_ps_drivers:rf_ctrl.c

 Line: 9348

 Logdata:

15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    ................
```

Figure 32. Example output of the AT+XLOG command

7.3 Vendor-specific AT commands

After examining the extracted list of available AT commands, we decided to look more closely at the mechanisms used to process them. We discovered that, in addition to the special characters "+", "^" or "%", the command interpreter also accepted "@". Upon receiving that character, the command handler's lookup mechanism behaved differently. Further analysis revealed that the modem supports many vendor-specific AT commands that are registered in the OS at the initialization stage (Figure 33).

kaspersky

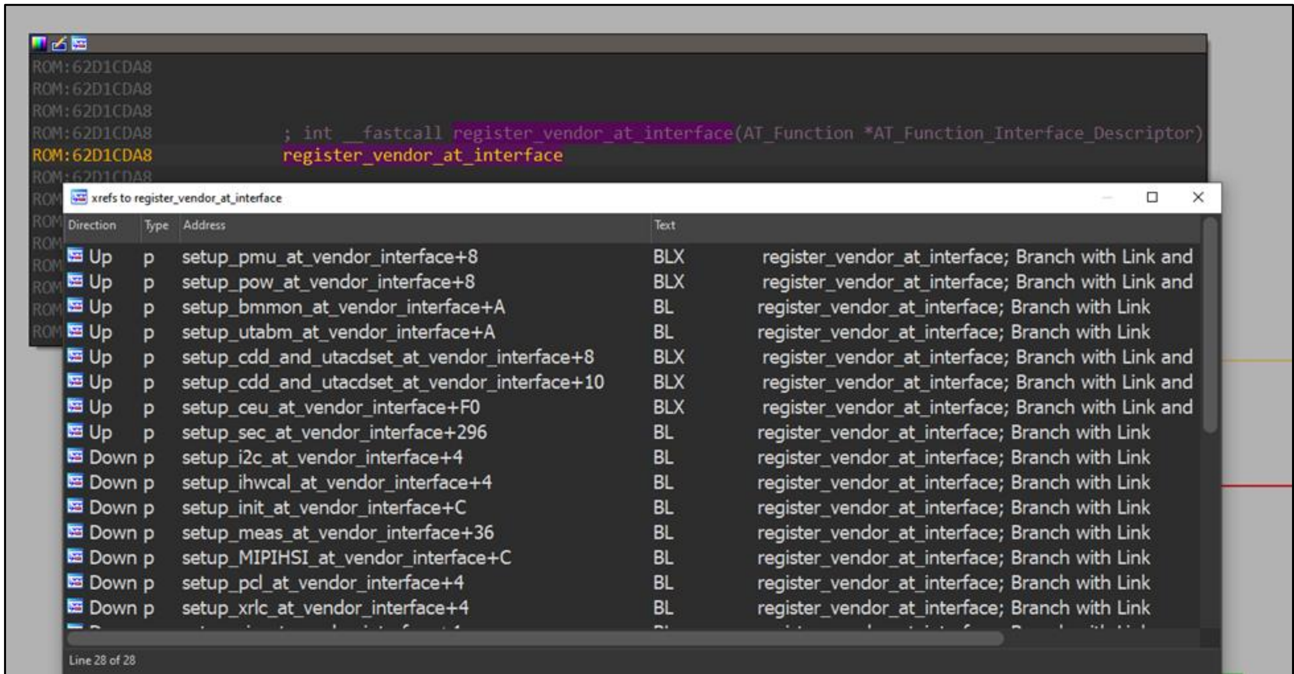


Figure 33. Registration of vendor AT commands

Registration consists of simply copying static data into a separate memory area (Figure 34). When these commands are processed later, this memory is accessed to find a suitable handler.

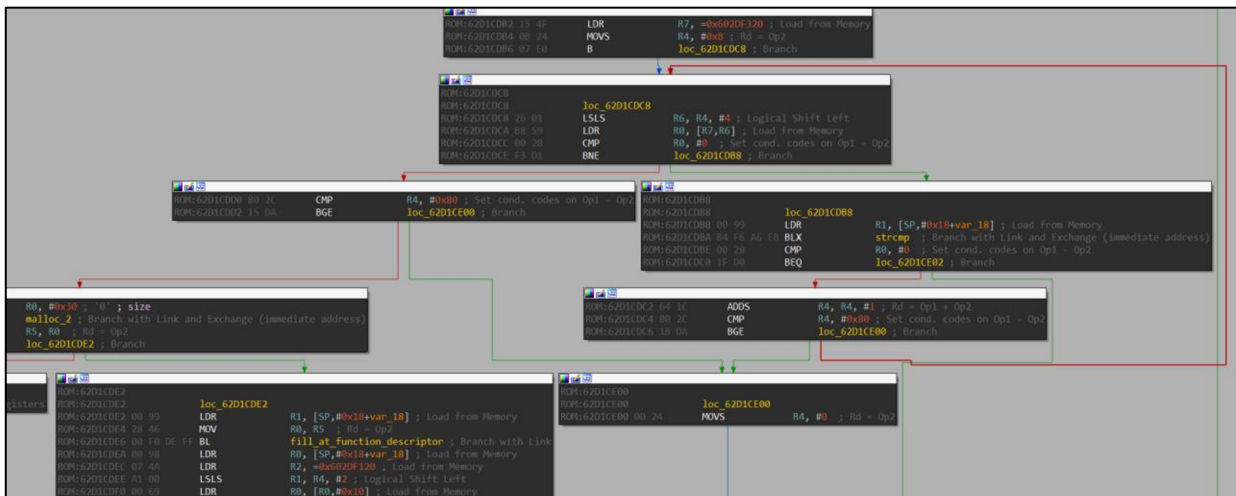


Figure 34. Registration of vendor AT commands

An analysis of the previously copied data showed that it contained a detailed description of each command (Figure 35). For example, we were able to find out the name of a command, its purpose, as well as the list of functions it supports and even the version of its implementation.

```
aUtabm          DCB "utabm",0           ; DATA XREF: setup_utabm_at_vendor_interface+8↑o
                                        ; ROM:off_62C8FE5C↑o
                DCB    0
                DCB    0
                DCB    0
                DCB    0
                DCB    0
                DCB    0
                DCD aUtaBmDebugInte     ; "UTA BM debug interface"
                DCD unk_62EFF8D0
                DCD off_62EFF770        ; "open_battery"
                DCD unk_600C39E4
                DCD sub_62C9100C+1
                DCD nullsub_342+1
a10001          DCB "1.00.01",0
                DCB    1
```

Figure 35. Copied data during command registration

The "SEC Security Interface" command looked particularly promising (Figure 36). It seemed to be exactly the kind of command that might be used to interact with the modem at the OS level.

```
DCB "sec",0               ; DATA XREF: setup_sec_at_vendor_interface+294↑o
                          ; setup_sec_at_vendor_interface:off_62D02B30↑o
DCD 0
DCD 0
DCD aSecSecurityInt       ; "SEC Security Interface"
DCD aQn                   ; "QN"
DCD off_62FFF5F8          ; "sec_format"
DCD unk_61A33D7C
DCD func_ata_sec_init+1
DCD func_ata_sec_kill_0+1
DCB "0.00.01",0
DCB    1
DCB    1
DCB    1
DCB    1
DCB    2
DCB    0
DCB    0
DCB    0
```

Figure 36. sec command

The features supported by this command, shown in Figure 37, also reinforced this assumption.

```
DCD aSecFormat            ; DATA XREF: ROM:62FFF684↓o
                          ; "sec_format"
DCD off_62FFF500
DCD aImeiRead             ; "imei_read"
DCD off_62FFF530
DCD aImeiLabel            ; "imei_label"
DCD off_62FFF540
DCD aCodeVerify           ; "code_verify"
DCD off_62FFF510
DCD aCodeClear            ; "code_clear"
DCD off_62FFF520
DCD aHwDetails            ; "hw_details"
DCD off_62FFF550
DCD aStatusInfo           ; "status_info"
DCD off_62FFF560
DCD aGetInfo              ; "get_info"
DCD off_62FFF570
DCD aStateInfo            ; "state_info"
DCD off_62FFF580
DCD aFlashIo              ; "flash_io"
DCD off_62FFF590
DCD aPtestGeneric         ; "ptest_generic"
DCD off_62FFF5A0
DCD aModuleTest           ; "module_test"
DCD off_62FFF5B0
DCD aFusScript            ; "fus_script"
DCD off_62FFF5C0
DCD aFusAttach            ; "fus_attach"
DCD off_62FFF5D0
DCD 0
```

Figure 37. Functions supported by the sec command

We also discovered that the format of vendor AT commands differs from the usual one. By analyzing their handler code, we determined that the general format of a vendor AT command is:

```
at@tag:(param1, ..., paramN);
```

The way these AT commands are called looks similar to calling a function in an interpreted scripting language. We discovered some useful commands supported by the corresponding interpreter:

- *at@help* - display the command help
- *at@E?* - get information about the execution error of the last command
- *at@*:?* - get a list of all available commands
- *at@tag:$*?* - get a list of all available functions supported by the command

Additionally, we found complete scripts for this interpreter within the OS code (Figure 38).

```
at_C_cmd_exec(0, -1, "getif:send_dsp_cs4_ber_cmd(1)", v22);
at_C_cmd_exec(0, -1, "gticom:delay(10)", v23);
at_C_cmd_exec(0, -1, "while(nof_arfcn_counter<nof_arfcn)", v24);
at_C_cmd_exec(0, -1, unk_62D16474, v25);
at_C_cmd_exec(0, -1, unk_62D16488, v26);
at_C_cmd_exec(0, -1, "gticom:delay(51)", v27);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:rx_sync_ver.flag?", v28);
at_C_cmd_exec(0, -1, "if(sync_result==1)", v29);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:sync_cnf_params.lfcb_result?", v30);
at_C_cmd_exec(0, -1, "if(sync_result<3)", v31);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:sync_cnf_params.sb_only_result?", v32);
at_C_cmd_exec(0, -1, "if(sync_result==1)", v33);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:sync_cnf_params.sb_only_data[0]?", v34);
at_C_cmd_exec(0, -1, "if(sync_result==0)", v35);
at_C_cmd_exec(0, -1, "gticom:counter=590", v36);
at_C_cmd_exec(0, -1, "else", v37);
at_C_cmd_exec(0, -1, "getif:start_mph_deactivate_req()", v38);
at_C_cmd_exec(0, -1, "gticom:delay(100)", v39);
at_C_cmd_exec(0, -1, "print(\"ERROR: LIVE NETWORK HAS BEEN FOUND \")", v40);
at_C_cmd_exec(0, -1, "endif", v41);
at_C_cmd_exec(0, -1, "endif", v42);
at_C_cmd_exec(0, -1, "endif", v43);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:sync_cnf_params.lfcb_result?", v44);
at_C_cmd_exec(0, -1, "if(sync_result==3)", v45);
at_C_cmd_exec(0, -1, "gticom:sync_result=getif:sync_cnf_params.sb_data[0]?", v46);
at_C_cmd_exec(0, -1, "if(sync_result==0)", v47);
at_C_cmd_exec(0, -1, "gticom:counter=590", v48);
at_C_cmd_exec(0, -1, "else", v49);
at_C_cmd_exec(0, -1, "getif:start_mph_deactivate_req()", v50);
at_C_cmd_exec(0, -1, "gticom:delay(100)", v51);
at_C_cmd_exec(0, -1, "print(\"ERROR: LIVE NETWORK HAS BEEN FOUND \")", v52);
at_C_cmd_exec(0, -1, "endif", v53);
at_C_cmd_exec(0, -1, "endif", v54);
at_C_cmd_exec(0, -1, "endif", v55);
at_C_cmd_exec(0, -1, "gticom:counter=counter+1", v56);
at_C_cmd_exec(0, -1, "endwhile", v57);
```

Figure 38. Scripts for the interpreter

The final step was to identify the parameters of the functions discovered in the *sec* command. This posed no issue, as the complete description of these commands were contained in the binary image of the OS (Figure 39).

```
off_62FFF540    DCD sub_62CF7FA8+1       ; DATA XREF: ROM:62FFF60C↓o
                DCD aLuS16               ; "%lu ...%s[16]"
                DCD aActionLabel         ; "action label"
                DCD aImeiLabelInter      ; "IMEI label interface. Use action=0 for "...
off_62FFF550    DCD func_ata_hw_details+1
                                         ; DATA XREF: ROM:62FFF624↓o
                DCD unk_63034BD2
                DCD unk_63034BD2
                DCD aReadsOutTheHwD      ; "Reads out the HW details."
off_62FFF560    DCD sub_62CF88AC+1       ; DATA XREF: ROM:62FFF62C↓o
                DCD aLu0                 ; "%lu=0"
                DCD aFormat_20           ; "format"
                DCD aReadsOutTheHwD_0    ; "Reads out the HW details."
off_62FFF570    DCD sub_62CF7C90+1       ; DATA XREF: ROM:62FFF634↓o
                DCD aS20                 ; "%s[20]"
                DCD aStringId            ; "string_id"
                DCD aReadsOutInfoRe      ; "Reads out info related to string id."
off_62FFF580    DCD sub_62CF8704+1       ; DATA XREF: ROM:62FFF63C↓o
                DCD unk_63034C4A
                DCD unk_63034C4A
                DCD aReadsOutSecuri      ; "Reads out security state info."
off_62FFF590    DCD func_ata_flash_io+1  ; DATA XREF: ROM:62FFF644↓o
                DCD aLuLuQu20LuLu        ; "%lu %lu ...*%&qu[20] %lu=# %lu"
                DCD aActionBlockidD      ; "action blockid data length"
                DCD aFlashIoInterfa      ; "Flash IO interface.\r\n\taction (0)=Rea"...
off_62FFF5A0    DCD func_ata_ptest_generic+1
                                         ; DATA XREF: ROM:62FFF64C↓o
                DCD aLuLuQu20Lu          ; "%lu %lu *%&qu[20] %lu=#"
                DCD aOpcodeLengthDa      ; "opcode length data"
                DCD aRunThePtestGen      ; "Run the ptest generic command."
off_62FFF5B0    DCD func_ata_module_test+1
                                         ; DATA XREF: ROM:62FFF654↓o
                DCD aLuLuQu20Lu_0        ; "%lu %lu *%&qu[20] %lu=#"
                DCD aOpcodeLengthDa_0    ; "opcode length data"
                DCD aRunTheModuleTe      ; "Run the module test command."
```

Figure 39. Full description of the commands available via the help command

For our initial test of the identified vendor command interface (Figure 40), we chose to use the *state_info* function, which was not expected to have any impact on the modem's functionality.

```
at@sec:state_info()
b_sys_tkt_testif = 0x0000
b_sys_tkt_bootcore = 0x0000
b_sys_tkt_secmodule = 0x0000
b_imei_data = 0x0000
b_sim_tkt_no = 0x0000
b_sim_tkt_ns = 0x0000
b_sim_tkt_sp = 0x0000
b_sim_tkt_cp = 0x0000
b_sim_tkt_sm = 0x0000
b_simlock_data = 0x0000
b_mid_certificate = 0x0002
s_valid_system_ticket = 0x0001
s_virgin_mode = 0x0001
result_cause = 0
```

Figure 40. state_info function

Then, we decided to use the *hw_details* function to get information about the modem hardware. However, we received an error instead of the expected modem hardware details (Figure 41).

```
At@sec:hw_details()
result_cause = 11
```

Figure 41. Error when executing the hw_details function

Further analysis showed that many of the functions available through the "*sec*" command verify the current level of user privileges. Figure 42 shows an example of this check for the *hw_details* function.

```
if ( v16[0] && a3 )
{
  if ( func_ata_switch_process(func_ata_hw_details, a1, v16, a3) )
    goto LABEL_5;
  v7 = func_ata_approve_access(1u);
  if ( !v7 )
  {
    v14 = v9;
    v7 = sub_62CFD090(v8, v9);
    if ( !v7 )
    {
      v13 = &v10;
      v7 = sub_62C2210C() != 0;
      if ( !v7 )
      {
        v16[0] = sub_62CF8368(v16[0], "hwid_bb", unk_61A33D74, (int)v8, 16u);
        v16[0] = sub_62CF8368(v16[0], "hwid_fc", unk_61A33D74, (int)v14, 16u);
        v16[0] = sub_62CF8368(v16[0], "hash_gpuk", unk_61A33D74, (int)v13, 20u);
        v16[0] = sub_62CF8368(v16[0], "hash_mpuk", unk_61A33D74, (int)v11, 20u);
        v16[0] = sub_62CF8368(v16[0], "hash_spuk", unk_61A33D74, (int)v12, 20u);
      }
    }
  }
  v16[0] = logging_to_user_console(v16[0], "result_cause = %hu", v7);
  goto LABEL_4;
}
```

Figure 42. Privilege level checks for hw_details function

The privilege level depends on the modem's current mode. Switching the modem to another mode requires a special vendor key (Figure 43) that is tied to the modem's hardware data: its model number, IMEI, and so on.

```
if ( func_glob_sec_init() )
{
  logging_1(1, "global service mode is SEC_SERVICE_FACTORY Access level: sec module access", 0);
  *sec_level = 3;                          // //here is unlocked sec at cmd level
}
else
{
  *sec_level = 0;
  v2 = SEC_operate_ticket_key(0, 0, 0, 0, 0, 255, sec_level);
  v3 = v2;
  if ( v2 && v2 != 6 )
  {
    *sec_level = 0;
    goto LABEL_7;
  }
}
```

Figure 43. Using a special vendor key

Due to the time-consuming nature of calculating the value of this key, the *sec* command proved ineffective for testing the confidentiality or integrity of the OS. Accordingly, we decided to explore other available vendor commands. Our attention was drawn to the *xl1* command (Figure 44).

```
DCD aXl1TraceInterf      ; "XL1 trace interface"
DCD unk_61F03620
DCD off_62F88D54         ; "dsptrace_start"
DCD dword_62F88E64
DCD sub_62ECE23C+1
DCD nullsub_340+1
DCD a10000_1             ; "1.00.00"
DCB    1
DCB    2
DCB    2
DCB    1
DCB    2
DCB    0
DCB    0
DCB    0
```

Figure 44. xl1 command

This command had functions with promising names (Figure 45).

```
DCD aReadMsErrorLog     ; "read_ms_error_log"
DCD off_62F889C8
DCD aSetRfAdjustMod_0   ; "set_rf_adjust_mode"
DCD off_62F889D8
DCD aGetRfAdjustMod_0   ; "get_rf_adjust_mode"
DCD off_62F889E8
DCD aPsvon              ; "psvon"
DCD off_62F889F8
DCD aPsvoff             ; "psvoff"
DCD off_62F88A08
DCD a2MemRd             ; "@2:mem_rd"
DCD off_62F88A78
DCD a2MemRdb            ; "@2:mem_rdb"
DCD off_62F88A88
DCD a2MemWr             ; "@2:mem_wr"
DCD off_62F88A98
DCD a2MemWrb            ; "@2:mem_wrb"
DCD off_62F88AA8
DCD aSetAthashMode      ; "set_athash_mode"
DCD off_62F88A28
DCD aSwReset            ; "sw_reset"
DCD off_62F88A38
DCD aSetStartupMode_0   ; "set_startup_mode"
DCD off_62F88A48
```

Figure 45. Functions of the xl1 command

The format of these functions (Figure 46) and their descriptions appeared to be related to reading and writing binary data in memory.

```
DCD sub_62ECE264+1        ; DATA XREF: ROM:62F88E08↓o
DCD aUHuHu0               ; "%u %hu %hu=0"
DCD aAddressLen           ; "address, len"
DCD aReadMemoryAddr       ; "read memory address (long) of specified"...
```

Figure 46. Format of function parameters

Our hypothesis was confirmed by analyzing their implementation (Figure 47). These were commands for reading/writing RAM. They were not removed from the production firmware.

```
v5 = a2;
if ( !*(_WORD *)(a3 + 4) )
  goto LABEL_4;
v7 = *(unsigned __int8 **)a3;
v8 = 0;
v11 = *(unsigned __int16 *)(a3 + 4);
while ( v8 < v11 )
{
  if ( !v5 )
    goto LABEL_29;
  if ( !(v8 << 28) && v5 > a2 )
  {
    sub_62D2B264(a1, a2, v5 - a2);
    v5 = a2;
  }
  if ( !(v8 << 30) )
    v5 = logging_to_user_console(v5, "\r\n%08lx: ", v7);
  v5 = logging_to_user_console(v5, "%08lx ", *(_DWORD *)v7);
  ++v8;
  v7 += 4;
}
```

Figure 47. Memory read command

Using the *at@help* command, we requested the modem's help output to check whether the *xl1* command was available. The output is shown in Figure 48.

```
gticom       Common Test Control Interface v.0.0.2
xl1          XL1 trace interface v.1.00.00
ver          Ver test interface v.01.00.0
pmu          PMU API AT test interface v.00.00.0
pow          POW API AT test interface v.00.00.0
ts           time services test interface v.01.00.0
meas         meas debug interface v.1.00.00
utasensor    UTA SENSOR interface v.1.00.00
utabm        UTA BM debug interface v.1.00.01
init         Init test interface v.01.00.0
uicc         UICC GTI SUPPORT v.1.00.00
bmmon        BMMON interface v.1.00.00
cdd          CDD test interface v.2.00.00
utacdset     *DEPRECATED* please use at@cdd v.2.00.00
vsyscal      VSYS calibration interface v.1.00.00
ihwcal       IHW calibration interface v.1.00.00
tbatcal      TBAT calibration interface v.1.00.00
tpcbcal      TPCB calibration interface v.1.00.00


trfcal       TRF calibration interface v.1.00.00
tbbiccal     TBBIC calibration interface v.1.00.00
sec          SEC Security Interface v.0.00.01
usbmwtestfw USB Middleware - Test Framework v.0.00.03

OK
```

Figure 48. Help output with the xl1 command

Our target command appeared in the list. However, when we tried to execute the *xl1* command, the read and write functions were not executed due to their non-standard definition in the firmware code. They had a special " *@2*" prefix in their names. To figure out

the meaning of this prefix, we re-examined the AT command processing algorithm. Our goal was to determine the conditions under which a function called within a command might not be executed.

We discovered that functions within a vendor-specific AT command can start with the "@" symbol followed by a numerical value from 0 to 2. This value indicates the level of user privileges required to execute these commands. Our privilege level was hard-coded in the modem's OS code at the static address *0x600D0AD0*. Since it is not assigned based on software commands or special hardware settings, there is no legitimate way to change it in the current build.

The modem operates with three privilege levels: 0, 1 and 2. These levels not only restrict access to functions within vendor-specific commands, but also govern the availability of the commands themselves.

We found that the privilege levels required for outputting the help information for commands may differ from what is required for executing those commands. Information about the required privilege levels is stored in the descriptors of each command. Checks are performed whenever a vendor-specific AT command processing function is called (Figure 49).

```
v6 = *(unsigned __int8 *)(*((_DWORD *)&unk_602DF320 + 4 * current_tag_number_matched_from_input_low) + 0x2C);
if ( (v6 > 4 || ((unsigned __int16)word_62F88024[v6] & off_600D0AD0) == 0) && (*(_WORD *)v61 & 0x40) == 0 )
  return 0;
```

Figure 49. Checking the required privilege levels

Based on the extracted information about the configuration and location of settings for accessing vendor-specific AT commands and their functions, we managed to reconstruct the list of all vendor-specific commands implemented in the modem OS (Table 1).

Command name	Description from firmware	Help available?	Privilege level
ufr	UMTS RF	no	2
utif	UMTS test interface	no	2
getif	GSM EDGE test interface	no	2
gcal	2G RF driver test and calib. interface	no	2
ucal	UMTS calibration interface	no	2
speed	full speed test interface	no	2
prodif	Production Interface (prodif)	no	2
prodctrl	Production Control Interface (prodctrl)	no	2
xl1	XL1 trace interface v.1.00.00	yes	2
ver	Ver test interface v.01.00.0	yes	1
pmu	PMU API AT test interface v.00.00.0	yes	1
pow	POW API AT test interface v.00.00.0	yes	1
ts	time services test interface v.01.00.0	yes	1
meas	meas debug interface v.1.00.00	yes	1
utasensor	UTA SENSOR interface v.1.00.00	yes	1
utabm	UTA BM debug interface v.1.00.01	yes	1
init	Init test interface v.01.00.0	yes	1
uicc	UICC GTI SUPPORT v.1.00.00	yes	1

bmmon	BMMON interface v.1.00.00	yes	1
cdd	CDD test interface v.2.00.00	yes	1
utacdset	DEPRECATED please use at@cdd v.2.00.00	yes	1
vsyscal	VSYS calibration interface v.1.00.00	yes	1
ihwcal	IHW calibration interface v.1.00.00	yes	1
tbatcal	TBAT calibration interface v.1.00.00	yes	1
tpcbcal	TPCB calibration interface v.1.00.00	yes	1
trfcal	TRF calibration interface v.1.00.00	yes	1
tbbiccal	TBBIC calibration interface v.1.00.00	yes	1
sec	sec v.0.00.01	yes	1
usbmwtestfw	USB Middleware - Test Framework v.0.00.03	yes	1
ceu	CEU Test Interface	no	2
i2c	I2C interface	no	1
mipihsi	GTI for MIPI	no	2
pcl	pcl interface	no	1
xrlc	GPRS-RLC functions provided to GTI Interface	no	2
sic	SIC Interface	no	2

Table 1. List of vendor-specific commands

All the commands shown in the table require either level-2 privileges or level-1 privileges. Note that the *xl1* command is the only command whose functions require level-2 privileges to run, but its help output can be viewed by a user with level-1 privileges. That's why we can see this command in help output but are unable to execute the corresponding functions. Given the presence of the *xl1* command, we could completely bypass the key checking restrictions in the *sec* command's functions if we could use its functions for reading and writing memory functions. We conclude that privilege levels 1 and 2 correspond to "OEM" and "Vendor" privilege levels, respectively.

7.4 Fuzzing AT commands

We used a Raspberry Pi 3 to build our fuzzing stand (Figure 50). We communicated with the AT interface via USB, and were able to perform a hard reboot of the modem using the appropriate hardware pin.

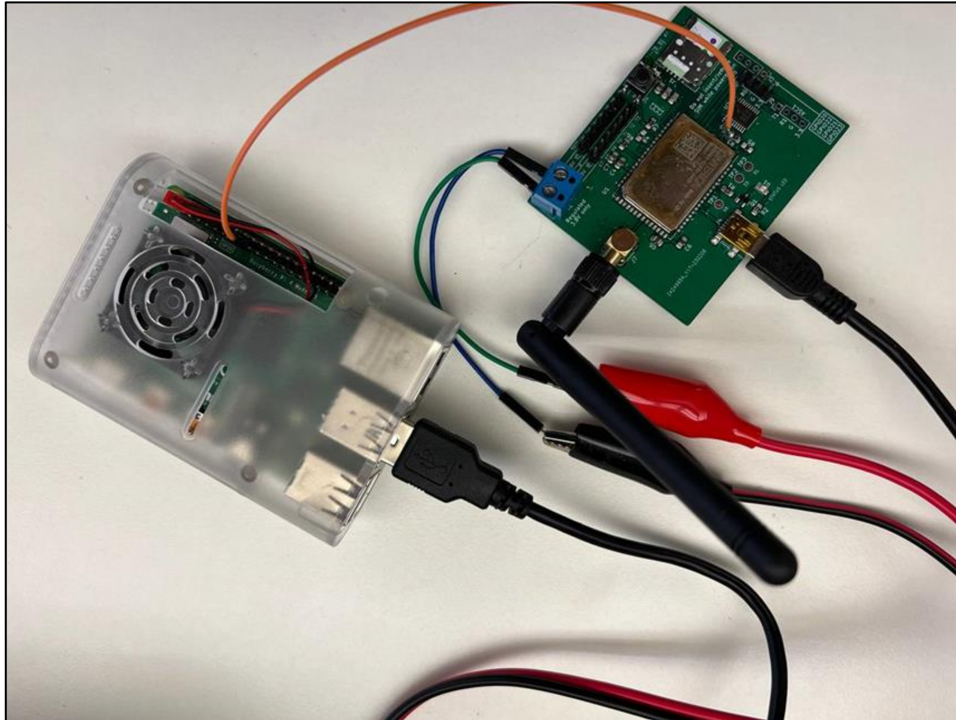


Figure 50. AT-command fuzzing stand

We built the fuzzing corpus using data obtained from our analysis of the AT commands. To develop a strategy for corpus generation and mutation, we identified AT command attributes that, if changed, could lead to execution errors:

1. Number of command parameters
2. Parameter type: string, number, binary value
3. Parameter value range

The mutation was performed by changing these parameters: creating or removing variables, altering the type of each variable, and modifying its range of values. The initial probability of these changes for each AT command was set at 1/2. However, this probability was not fixed; it varied depending on the results of a single corpus test of a particular AT command. After each corpus test, we evaluated the number of command execution errors on the modem as well as their execution time. The mutation probability for each attribute was adjusted up or down accordingly. As a result, each AT command had a unique probability that its attributes would change.

During the fuzzing process, we had to discern when the modem was in an operational state. Error detection was based on three conditions that allowed us to reliably identify the current state:

1. AT interface unavailable
2. No response from the AT interface
3. The last line read is not one of the following: *ERROR*, *CMD ERROR*, *NO CARRIER*, *OK*, *ABORTED*.

If any of these tests failed, it was assumed that an error had occurred in the execution of the AT command. The cause of the error could be extracted using the *AT+XLOG* command. All commands sent between two detected errors were collected in separate logs so that the

result could be repeated on the same data set. A hard reboot of the modem ensured that the previous error did not affect the test results.

After all the preparations, the fuzzing stand was switched on and left to run for several months. Many errors were recorded as the result of fuzzing. Most of these errors were software exceptions, which means that they were not exploitable. However, we were able to detect a stable memory access error when the *AT+XMUX* command was executed. Detailed analysis allowed us to discover a heap overflow vulnerability during the handling of this command.

The command takes 11 parameters as input. Each of these parameters is a single-byte number. Each received AT command goes through a conversion phase at the beginning of its handler where the input parameters are converted from a string to the required data type, which is a number in this case. The conversion is performed with help of a library function (Figure 51) that takes a pointer to a buffer containing a string with the AT command and the number of parameters as input. The third parameter returns the conversion result as an array of data representing the result. The software error can be found in this function.

```
int __fastcall xmux_set(int a1, int a2)
{
  int v2; // r4
  int v3; // r0
  unsigned int i; // r5
  int v5; // r0
  int v6; // r0
  int v8[11]; // [sp+4h] [bp-4Ch] BYREF
  int v9; // [sp+30h] [bp-20h] BYREF
  int v10; // [sp+34h] [bp-1Ch]
  int v11; // [sp+38h] [bp-18h]

  v10 = a1;
  v11 = a2;
  v2 = 0;
  if ( !atCmdParams_isNumericType(a2, 11, &v9) )
  {
```

Figure 51. Library function performing data conversion

The function itself is not supposed to convert a string with more than 11 parameters (Figure 52).

```
if ( !user_input_buffer )
  return 9;
if ( params_count > 11 )
  return 9;
```

Figure 52. Limit on the number of parameters

Each of these parameters must be limited in size by the intermediate buffer length. This buffer is used to perform the string-to-number conversion of each of the parameters

individually. Its size is hard coded in the function code. It is also equal to 11 bytes allocated on the heap (Figure 53).

```
if ( *user_input_buffer )
{
  v23 = cat_memalloc_ext(11);
  if ( v23 )
  {
```

Figure 53. Limit on the size of parameters

Inside the function there is a check that the number of parameters in the processed string does not exceed the expected number (Figure 54).

```
if ( v5 >= params_count )
{
  ATCmdParams_destroy(v22);
  free_0(v23);
  return 5;
}
if ( *user_input_buffer == ',' || !*user_input_buffer )
{
  if ( v24 == 1 )
  {
    if ( *(user_input_buffer - 1) == ',' )
    {
      v16 = &v25[2 * v8];
      *v16 = 0;
      v16[1] = 0;
      v24 = 0;
      v8 = (char)(v8 + 1);
      v5 = (char)(v5 + 1);
    }
    v17 = user_input_buffer;
  }
```

Figure 54. Checking for the expected number of parameters

However, the function does not check if the length of a parameter is greater than the size of the buffer allocated to convert it to a number (Figure 55). In this case, each parameter of the input string is copied until the "," delimiter is encountered or the end of the input string is reached.

```
do
{
  v10 = (unsigned __int8)*user_input_buffer;
  if ( *(_BYTE *)(*v9 + v10) == ' ' )
  {
    v11 = v3;
    ++user_input_buffer;
    v3 = (char)(v3 + 1);
    *(_BYTE *)(v23 + v11) = v10;
  }
  else
  {
    if ( v10 != '-' )
    {
      ATCmdParams_destroy(v22);
      free_0(v23);
      return 23;
    }
    ++user_input_buffer;
    *(_BYTE *)(v23 + v3) = '-';
    v3 = (char)(v3 + 1);
    if ( *(_BYTE *)(*v9 + (unsigned __int8)*user_input_buffer) != ' ' )
    {
      ATCmdParams_destroy(v22);
      free_0(v23);
      return 9;
    }
  }
}
while ( *user_input_buffer != ',' && *user_input_buffer );
*(_BYTE *)(v23 + v3) = 0;
v13 = &v25[2 * v8];
*(_QWORD *)v13 = ((__int64 (__fastcall *)(int))str2int)(v23);
```

Figure 55. Error in the parameter length check

The bytes of the input parameter are added to the same buffer. The first parameter of an AT command can overflow this buffer if its length exceeds 11 bytes.

Among all AT command handlers, this library function is used only in the *AT+XMUX* command handler. This command accepts only numbers as data, which are also checked during processing. Accordingly, we cannot exploit this error to achieve heap overflow with arbitrary binary data. We therefore decided to search for another option to implement a full-blown arbitrary code execution exploit.

7.5 SMS message processing

We decided to analyze the security of SMS message processing, starting from the moment the message is received. To achieve this, we needed to reconstruct the algorithm for processing SMS messages, tracing the process from receiving the raw message from the DSP to its final processing. Several processes[10] are involved in SMS message processing, as illustrated in Figure 56.

[10] In the ThreadX operating system (now Azure RTOS ThreadX), there is no concept of processes, instead, it operates on threads. However, since the OEM manufacturer uses the term "process" in the firmware we were studying, we will also use this term throughout the text of the article to maintain consistent terminology.

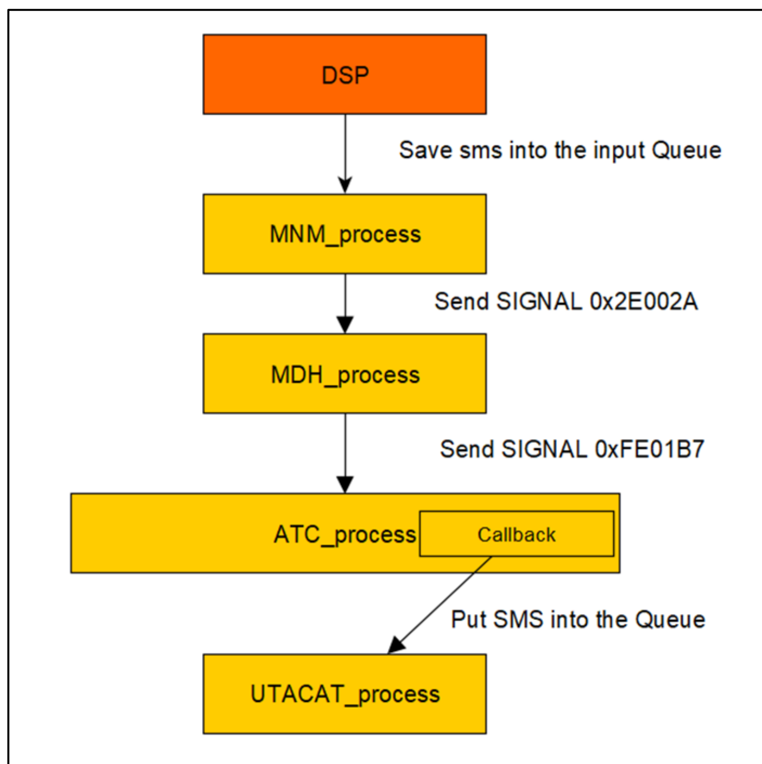


Figure 56. Processes in SMS handling

The first process to handle a received SMS message is the *MNM* process. It is responsible for checking whether the received SMS message conforms to the SMS PDU format. If the body of the received SMS message complies with the standard, it is sent to the next process (Figure 57). It is important to note that this process, shown in Figure 56, not only is responsible for processing SMS messages, but also participates in the processing pipeline of various messages and events coming from the mobile network.

```
case 0xE:
  v102 = *(_DWORD *)(v2 + 20);
  j_call_at_func(v225, (char *)(v2 + 24), 0xB2u, (int)v197);
  v73 = &v105;
  v74 = v102;
  sub_63299BC0(v200);
  if ( v105 < 0 )
    sub_631577B2(1524, "..\\..\\umts-build\\HW_SIC\\sdl\\text\\mnm.c", 4);
  v194 = sub_6332DFD8((int)v225);
  if ( !v194 )                          // here if sms deliver or reserved :)
  {
    result[9 * v105 + 0x137] = 0;
    v20 = j_parse_sms_header((int)v225, (int)&v156);
    v107 = v20;
    if ( v20 != 0x21D )                 // operate error
      goto LABEL_58;
    j_memcopy(v223, v157, 0x2Au);
    v74 = 0;
    v73 = (int *)v102;
    LOWORD(v75) = v158;
    goto LABEL_64;
  }
  if ( v194 == 1 )
  {
    result[9 * v105 + 311] = 2;
    v22 = sub_6332DFE8((int)v225, (int)&v159);
    v107 = v22;
    if ( v22 != 0x21D )
    {
_58:
      v21 = j_alloc_sig_loc(0x1Cu, "mn:mnm.c", 1442);
      *(_DWORD *)(v21 + 12) = v67;
      *(_BYTE *)(v21 + 16) = -1;
      *(_DWORD *)(v21 + 20) = result[9 * v105 + 310];
      *(_WORD *)(v21 + 24) = v107;
      *(_DWORD *)(v21 + 8) = 0x5F0006;
      return sub_63155CD0();
    }
    j_memcopy(v223, (char *)&v159 + 3, 0x2Au);
    v74 = 2;
    v73 = (int *)v102;
    LOWORD(v75) = 0;
_64:
    v76 = 0;
    v77 = v224;
    send_sdl_sms_event_signal(v200);
```

Check SMS structure

Send next process

Figure 57. Processing the body of a received SMS

In this case, the only interesting function for us is one that can be affected by modifying the SMS message content, i.e. the function responsible for checking the message structure. It contains simple checks to ensure that field values of the received SMS comply with expected values. Each of these checks may interrupt the processing of the received SMS. Among all the fields that can be modified, there is one that deserves attention, namely the OA field (Figure 58).

kaspersky

```
memcopy((_BYTE *)a2, sms_body_ptr, v8 + 1);   // here copy sca
pdu_type_offset = v29 + 1;                     // pdu type
v10 = (unsigned __int8)v29[1];
if ( v10 << 30 )
{
  if ( ~v10 << 30 )
    return 0x60;
}                                              // accept only pdu with sms-dekiver type or reserved
*(_BYTE *)(a2 + 42) = (v10 << 29 >> 31) + 1;  // mms bit
oa_offset = pdu_type_offset + 1;
*(_BYTE *)(a2 + 43) = *pdu_type_offset >> 7;
*(_BYTE *)(a2 + 44) = (*pdu_type_offset & 0x40) != 0;
*(_BYTE *)(a2 + 45) = (*pdu_type_offset & 0x20) != 0;
if ( (unsigned int)(pdu_type_offset + 1) >= sms_end_ptr )
  return 0x60;
if ( (unsigned int)&oa_offset[(((unsigned int)*oa_offset + 1) >> 1) + 1] >= sms_end_ptr )
  return 0x60;
v28 = a2 + 0x2E;
memSet((_BYTE *)(a2 + 0x2E), 0x2Au, 0xFF);
sub_62C6A860(oa_offset, (_BYTE *)(a2 + 0x2E), v25);
if ( (unsigned __int8)v25[0] > 0xCu )
  return 0x60;
v12 = &oa_offset[(unsigned __int8)v25[0]];
if ( (unsigned int)(v12 + 0xA) > sms_end_ptr )
  return 0x60;
v13 = *v12;                                    // pid
v14 = v12 + 1;                                 // dsc
*(_BYTE *)(a2 + 0x58) = v13;
v15 = v14 + 1;                                 // scts offset
*(_BYTE *)(a2 + 89) = *v14;
sub_62CBFDE4((_BYTE *)(a2 + 90), v14 + 1);
*(_DWORD *)(a2 + 100) = (unsigned __int8)v15[7];
v26 = v15 + 8;                                 // ud
sub_62DE5BB0(*(unsigned __int8 *)(v16 + 25), v25, v24, v23, v22, v21, v20);
v17 = *(_DWORD *)(a2 + 100);
v18 = (unsigned __int8)v17;
if ( v24[0] == 1 || !v24[0] )
  v18 = (unsigned __int8)(v17 - v17 / 8);
v27 = &v26[v18];
if ( (unsigned int)&v26[v18] > sms_end_ptr || *(_BYTE *)(a2 + 44) && (unsigned __int8)*v26 > v18 )
  return 0x60;
if ( v18 > 0x8C )
  return 0x60;
memSet((_BYTE *)(a2 + 104), 0x8Cu, 255);
memcopy((_BYTE *)(a2 + 104), v26, v18);
*(_BYTE *)(v30 + 1) = (_BYTE)v27 - v30 - 2;   // setup new sms body size
```

Store OA in structure of size 0xF4 at offset 0x2E

Check OA size after copy

Figure 58. OA field

According to the standard, it is assumed that the OA (Originating Address) field stores the number of the sender of the message (Figure 59). The length of this number must not exceed 0xC bytes, and there is a corresponding check in the code.

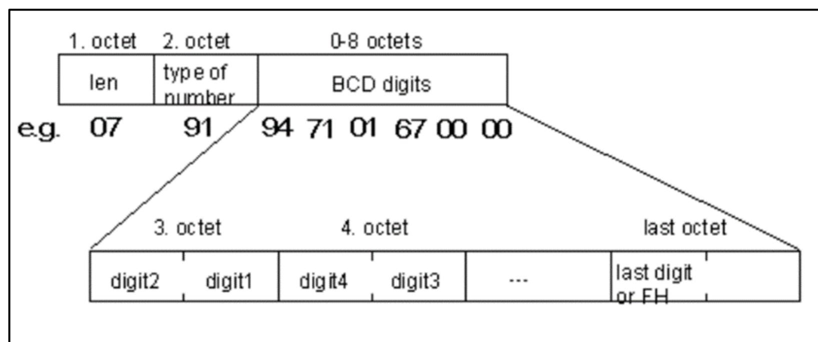


Figure 59. Contents of the OA field according to the standard

However, this check is performed only after the data from the message has been processed. There is no such check within the handler itself (Figure 60).

```
int __fastcall sub_62C6A860(char *a1, _BYTE *to, _BYTE *a3)
{
  int v6; // r0
  unsigned int v7; // r6
  char v8; // r0
  int v9; // r6
  int result; // r0
  int v11; // [sp+4h] [bp-1Ch] BYREF
  int v12; // [sp+8h] [bp-18h] BYREF

  v12 = 0;
  v11 = 0;
  sub_62D11258(a1[1], &v12, &v11);
  if ( (unsigned __int8)v12 != 5 )
    return sub_62C6A82C(a1, to, a3);
  v6 = (unsigned __int8)*a1;
  v7 = (unsigned int)(v6 + 1) >> 1;
  *a3 = v7 + 2;
  sub_62BD4DAC(v6 + 1, 0xEu);
  v9 = (unsigned __int8)(v8 + v7);
  memSet(to, 0x2Au, 0xFF);
  to[1] = a1[1];
  result = sub_62DE98EC(4, a1 + 2, v9, to + 2) + 1;
  *to = result;
  return result;
}
```

Buffer expects 0x2A bytes

But works with the number of bytes equal to the size in the OA header

Figure 60. No check of the length of the number of the message sender

Although the parsing of the message header is interrupted, it is possible to copy up to *0x80* bytes into a buffer that should not contain more than *0x2A* bytes. This is a software bug in the lowest-level driver that handles SMS messages. However, in this case, the error does not result in a vulnerability. The buffer with this copy error is part of the received SMS message descriptor structure with the size *0xF4*, and it is located almost at the beginning at offset *0x2A*. If the structure that stores the processed SMS message header had a different format, then overflow would be possible.

After all header fields have been checked, the received SMS is passed to the *MDH* process for further processing. This handoff is accomplished by sending the *0x2E002A* signal (Figure 61).

kaspersky

```
*(_BYTE *)very_big_sms_extend_structure[5] = 1;
if ( *(_BYTE *)(v2 + 1352) )
{
  *(_BYTE *)(v2 + 1356) = 1;
  v4 = *(_DWORD *)(v2 + 1372) + 1;
  *(_DWORD *)(v2 + 1372) = v4;
  *(_DWORD *)(v2 + 1360) = v4;
  v5 = very_big_sms_extend_structure[1];
  *(_DWORD *)(v2 + 1368) = v4;
  *(_DWORD *)(v2 + 1364) = v5;
  if ( v4 == 255 )
    *(_DWORD *)(v2 + 1372) = 0;
  v6 = j_alloc_sig_loc(0xDCu, "mn:mnm.c", 6640);
  *(_DWORD *)(v6 + 12) = *(_DWORD *)(v2 + 8);
  *(_BYTE *)(v6 + 16) = -1;
  *(_BYTE *)(v6 + 20) = *(_BYTE *)(v2 + 3880);
  *(_DWORD *)(v6 + 24) = *(_DWORD *)(v2 + 1368);
  j_call_at_func((_DWORD *)(v6 + 28), (char *)(v2 + 0xCA8), 0xB2u, v7);// call memcopy
  *(_DWORD *)(v6 + 208) = very_big_sms_extend_structure[2];
  v8 = 0x19FD;
  *(_BYTE *)(v6 + 212) = *((_BYTE *)very_big_sms_extend_structure + 12);
  *(_BYTE *)(v6 + 213) = *((_BYTE *)very_big_sms_extend_structure + 13);
  *(_DWORD *)(v6 + 216) = very_big_sms_extend_structure[4];
  v9 = v6;
}
else
{
  v10 = j_alloc_sig_loc(0xDCu, "mn:mnm.c", 6658);
  *(_DWORD *)(v10 + 12) = *(_DWORD *)(v2 + 8);
  *(_BYTE *)(v10 + 16) = -1;
  *(_BYTE *)(v10 + 20) = *(_BYTE *)(v2 + 3880);
  *(_DWORD *)(v10 + 24) = very_big_sms_extend_structure[1];
  j_call_at_func((_DWORD *)(v10 + 28), (char *)(v2 + 0xCA8), 0xB2u, v11);
  *(_DWORD *)(v10 + 208) = very_big_sms_extend_structure[2];
  v8 = 6671;
  *(_BYTE *)(v10 + 212) = *((_BYTE *)very_big_sms_extend_structure + 12);
  *(_BYTE *)(v10 + 213) = *((_BYTE *)very_big_sms_extend_structure + 13);
  *(_DWORD *)(v10 + 216) = very_big_sms_extend_structure[4];
  v9 = v10;
}
return j_send_sig_no_to_loc(v9, 0x2E002Au, "mn:mnm.c", v8);
```

Figure 61. Transmitting the message further on via signal 0x2E002A

This signal is handled by the *MDH* process (Figure 62). The SMS message is processed by the *ATC* process via the *0xFE01B7* signal (Figure 63).

```
if ( signal_code == 0x2E002A )
{
  sms_data_buffer = (char *)(sms_data_buffer_external_struct + 28);
  v16 = *(_DWORD *)(sms_data_buffer_external_struct + 24);
  v17 = sub_63380CB4(unk_61D9EE3F);
  if ( v17 )
    here_operate_incoming_sms_mdh(v17, v16, sms_data_buffer);
  return v5;
}
```

Figure 62. Signal processing by the MDH process

```
v5 = j_alloc_sig_loc(0xCCu, "dr:mdh_mmi_sms_handler.c", 98);
*(_DWORD *)(v5 + 12) = MEMORY[0x70844EA0];
*(_BYTE *)(v5 + 16) = -1;
*(_DWORD *)(v5 + 20) = a2;
j_memcopy((_BYTE *)(v5 + 24), sms_data_buffer, 0xB2u);
*(_DWORD *)(v5 + 8) = 0xFE01B7;
return send_sig_loc_ext(v5);
```

Figure 63. Passing the message to the ATC process via the 0xFE01B7 signal

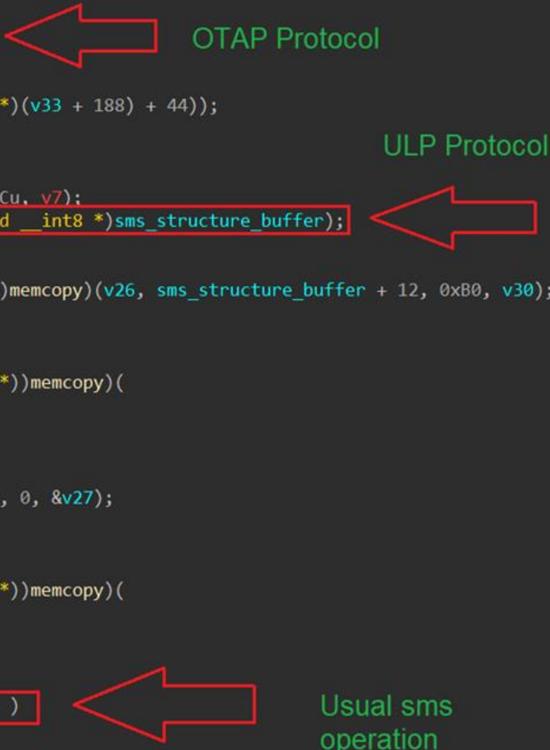
This signal is further processed in the callback handler of the *UTACAT* process, which *UTACAT* itself registers when it begins running. This callback puts the SMS message into the message queue of the *UTACAT* process (Figure 64).

```
v9 = a4;
result = cat_memalloc_ext(0xC8, (int)a2, a3, a4);
v8 = result;
if ( result )
{
  *(_BYTE *)result = 5;
  *(_DWORD *)(result + 4) = a1;
  *(_WORD *)(result + 8) = 0x98;
  programm_memcopy((_DWORD *)(result + 12), a2, 0xBCu, v7);
  v9 = v8;
  return send_IPC_to_thread(1u, (int)&v9);
}
return result;
```

Figure 64. Signal processing in the callback handler of the "UTACAT" process

Next, *UTACAT* processes the body of the received SMS message (Figure 65). In this case, right from the start, there is an unconditional check to ensure that the received SMS messages comply with the OTAP and ULP protocols[11]. Besides these two protocols, the modem only processes ordinary SMS messages.

[11] https://www.openmobilealliance.org/release/supl/V1_0-20070615-A/OMA-TS-ULP-V1_0-20070615-A.pdf



```
int __fastcall OperateSMS(int a1, int a2, char *sms_structure_buffer)
{
  v32 = a1;
  v33 = a2;
  v34 = sms_structure_buffer;
  v27 = 0;
  v4 = (_DWORD *)sub_62EB1E7A(a2);
  sub_62CA3460(*(_DWORD *)(v33 + 200), 2u);
  sub_62CA3460(*(_DWORD *)(v33 + 200), 0);
  v21 = sub_62CA3460(*(_DWORD *)(v33 + 200), 2u);
  v5 = sub_62CA3460(*(_DWORD *)(v33 + 200), 0);
  result = j_OTAPSmsOperate(sms_structure_buffer);
  if ( result )
    return result;
  result = sub_62CA8832(*(unsigned __int8 *)(*(_DWORD *)(v33 + 188) + 44));
  v8 = result;
  if ( !sms_structure_buffer )
    return result;
  programm_memcopy(v4 + 227, sms_structure_buffer, 0xBCu, v7);
  v24 = j_Pos__Cat_GPSMngr_handleSUPLsms(v33, (unsigned __int8 *)sms_structure_buffer);
  v25[4] = sms_structure_buffer[11];
  sms_header = sms_structure_buffer + 12;
  ((void (__fastcall *)(_BYTE *, char *, int, _BYTE *))memcopy)(v26, sms_structure_buffer + 12, 0xB0, v30);
  memFill(&sms_data_buffer, 180u);
  LOBYTE(sms_data_buffer) = 3;
  BYTE1(sms_data_buffer) = sms_structure_buffer[11];
  ((void (__fastcall *)(char *, char *, _DWORD, _BYTE *))memcopy)(
    (char *)&sms_data_buffer + 2,
    sms_header,
    (unsigned __int8)sms_structure_buffer[11],
    v30);
  sub_62CA555C((unsigned __int8 *)&sms_data_buffer + 2, 0, &v27);
  *((_BYTE *)v4 + 1956) = v27;
  *((_BYTE *)v4 + 1096) = 3;
  *((_BYTE *)v4 + 1097) = sms_structure_buffer[11];
  ((void (__fastcall *)(char *, char *, _DWORD, _BYTE *))memcopy)(
    (char *)v4 + 1098,
    sms_header,
    (unsigned __int8)sms_structure_buffer[11],
    v30);
  if ( sms_data_unpack((int)&sms_data_buffer, v28, v8) )
  {
    v9 = 24;
    v10 = v32;
    return Save_sms_to_Mem(v4, (unsigned __int8 *)sms_structure_buffer, v9, v10);
```


Figure 65. SMS message handling by the UTACAT process

7.6 OTAP protocol analysis

In the OTAP protocol, data is transmitted using SMS messages that have special values for the Class and PID fields (Figure 66).

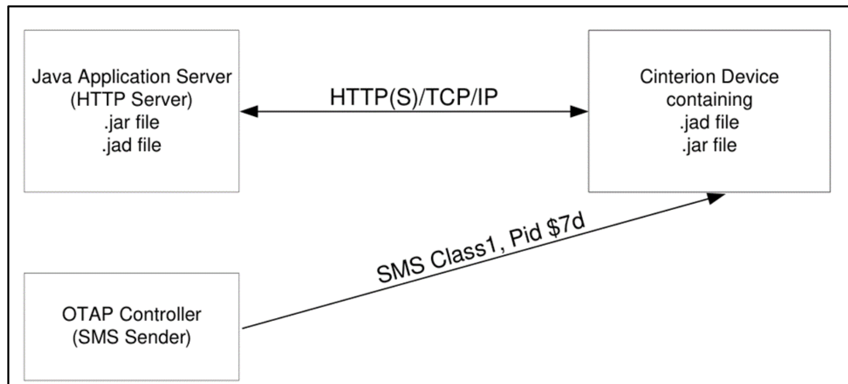


Figure 66. Data transfer in the OTAP protocol

OTAP messages are ASCII text contained within the SMS message body. An example of such a message is shown in Figure 67.

```
Delete operation:
    OTAP_IMPNG
    PWD:secret
    JADURL:http://www.greatcompany.com/coolapps/mega.jad
    START:delete
```

Figure 67. Example of an OTAP message

The initial processing of the received message takes place in the *UTACAT* process. This process is responsible for decoding the received SMS message and sending it to the process that handles the OTAP protocol.

Only two functions responsible for decoding the OTAP protocol can be affected. The first is the function that converts the received message to text format (Figure 68).

```
if ( convert_to_ascii((int)v20, v21, (int)v13 + 1, 160, &v12) )
{
  JavaLog(27, (int)"[ERR][%s(L:1321)]leave,Convert to ASCII failed", &aOtapDodelete[0x1E]);
  return 0;
}
```

Figure 68. Function for converting the received message to text format

The second is used to check that the OTAP SMS ID is present in the message header (Figure 69).

```
if ( LOBYTE(v13[0]) < 0xAu || str_cmp((char *)v13 + 1, "OTAP_IMPNG", 10) )
{
```

Figure 69. Checking whether the OTAP SMS ID is present in the message header

During the research, we did not find any errors in the implementation of these two functions. Moreover, the transmission protocol itself is trivial, so we proceeded to analyze the functions that handle the processing of the received message.

The OTAP message handler is located in the native MIDP library implementation of the *JVM* process, i.e. the Java interpreter. The *JVM* process itself greets the researcher with a wonderful message (Figure 70).

```
sub_634F4D54(v13, byte_61A6E940, 7);
JavaLog(0, (int)"[%s(L:1140)]SOMETHING WONDERFUL HAS HAPPENED!!\r\n", &loc_637BE68A);
v32 = sub_62C5AF34(1643134216);
```

Figure 70. Surprise log message

And it instantly reveals all the secrets about the location of MIDlets and other system files and folders (Figure 71).

```
sub_62BD1450("/sys", (int *)"/sys", 4);
sub_62BD1450(aSys_4, (int *)"/sys/", 5);
sub_62BD1450(&aSys_4[255], (int *)"/sys/.cinterion.internal/java", 29);
sub_62BD1450(&aSys_4[765], (int *)"A:/.cinterion.internal/java", 27);
sub_62BD1450(v41, (int *)"/sys/.cinterion.internal/java/_main.ks", 38);
sub_62C458C8("/sys/.cinterion.internal");
sub_62C458C8("/sys/.cinterion.service");
sub_62C458C8("/sys/.cinterion.service/SLAE");
sub_62C458C8("/sys/.cinterion.internal/java");
sub_62C458C8("/sys/.cinterion.internal/java/storage");
sub_6368E77C((int)"system.storage_root", (int)"/sys/.cinterion.internal/java", 1, 0);
sub_6368E77C((int)"profiler.filename", (int)"/sys/graph.prf", 1, 0);
sub_6368E77C((int)"system.default_storage", (int)"/sys/.cinterion.internal/java/storage", 1, 0);
```

Figure 71. Storage data for MIDlets and other system files and folders

The code for handling a received OTAP message has quite a lot of comments. This allowed us to quickly determine that if OTAP was not previously activated by executing the *AT^SJOTAP* command, then the received message would not be processed (Figure 72). Activation involves creating a special OTAP settings file in the UFS. Since our goal was to analyze protocol security without putting the system into special operating conditions, we decided to proceed with analyzing the implementation of the ULP protocol.

```
JavaLog(27, (int)"[VBS][%s(L:1592)]enter.", "OtapSMSin2");
if ( !a1 )
  return JavaLog(27, (int)"[ERR][%s(L:1596)]leave, pdata == NULL.", "OtapSMSin2");
memcopy(v8, a1, 161u);
java_mem_free((int)a1, (int)"OtapSMSin2", 0x641);
if ( unk_600DE5C0 )
  return JavaLog(27, (int)"[WRN][%s(L:1606)]leave, OTAP already in progress -> ignoring", "OtapSMSin2");
JavaLog(27, (int)"[VBS][%s(L:374)]enter.", "isOtapFilePresent");
v3 = (int)sub_62BD14C0(aCinterionInter_1);
memFill(aCinterionInter_1, v3);
strcopy(0x61A7E788, 0x6181CFEB);
strcopy(0x61A7E788, 0x600DE5D0);
if ( isFilePresent ("/.cinterion.internal/java/Otap_AtParams.bin", (int)v7) < 0 )
{
  JavaLog(27, (int)"[ERR][%s(L:383)]retval < 0, leave.", "isOtapFilePresent");
  return JavaLog(27, (int)"[ERR][%s(L:1613)]leave, OTAP has never been configured -> ignoring.", "OtapSMSin2");
}
if ( (v7[2] & 0x200) != 0 )
{
  JavaLog(27, (int)"[ERR][%s(L:389)]UTA_FS_ATTR_DIR, leave.", "isOtapFilePresent");
  return JavaLog(27, (int)"[ERR][%s(L:1613)]leave, OTAP has never been configured -> ignoring.", "OtapSMSin2");
}
JavaLog(27, (int)"[INF][%s(L:394)]File Present, leave.", "isOtapFilePresent");
JavaLog(27, (int)"[INF][%s(L:1617)]before OTAP_SmsProcess\n", "OtapSMSin2");
if ( OTAP_SmsProcess(v8, byte_600DE5B0) )
{
  JavaLog(27, (int)"[INF][%s(L:1621)]after OTAP_SmsProcess, otapOp = %d\n", "OtapSMSin2", byte_600DE5B0[0]);
  if ( byte_600DE5B0[0] == 1 )
  {
```

Figure 72. OTAP activation check

7.7 ULP protocol analysis

In addition to the remote provisioning of MIDlets via SMS messages using the OTAP protocol, the modem also offers geopositioning capabilities using the SUPL subsystem[12]. This subsystem facilitates the exchange of special messages between H-SLP (Home SUPL

[12] https://www.openmobilealliance.org/release/SUPL/V2_0-20120417-A/OMA-AD-SUPL-V2_0-20120417-A.pdf

Location Platform) and SET (SUPL Enabled Terminal). According to the specification, the modem functions as a SET object. An example of this interaction is shown in Figure 73.

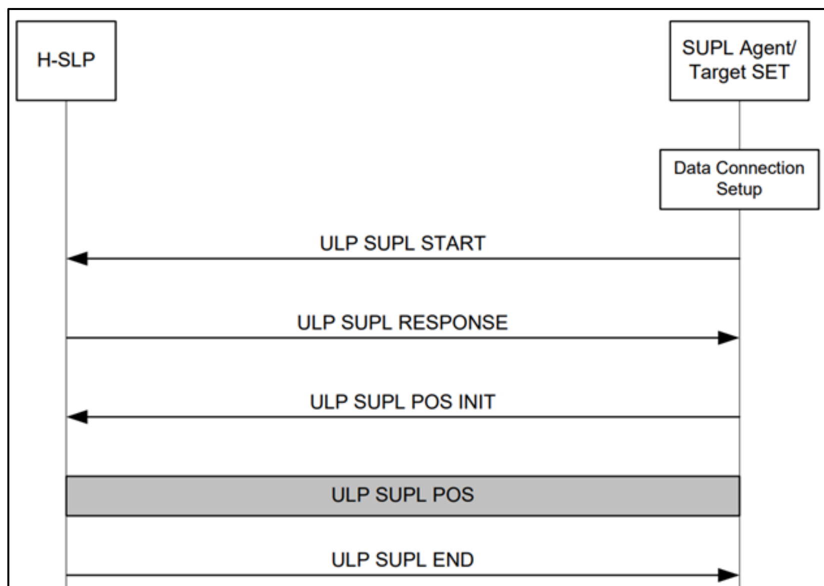


Figure 73. Interaction via the ULP protocol

Messages are exchanged using the ULP binary protocol. In this protocol, data is transmitted in the GSM network via PUSH messages using WAP stack protocols[13]. A typical ULP message is illustrated by the SUPL INIT message (Figure 74).

[13] UserPlane Location Protocol, Open Mobile Alliance™: https://www.openmobilealliance.org/release/SUPL/V2_0_5-20191028-A/OMA-TS-ULP-V2_0_5-20191028-A.pdf

Field	Reference	Size	Type	Value
WSP PDU Header				
TID		1	Octet	—
PDU Type		1	Octet	0x06
Push Header Length		1	Octet	(varies)
content type		(depends on *Value* chosen)	Octet	(varies)
Push Header				
x-wap-application-id		1	Octet	0xAF
x-application-Id-field		(depends on *Value* chosen)	Octet	(varies)
Push Content				
SUPL INIT Message		N	Octet	—

Figure 74. SUPL INIT message

The ULP protocol includes the ability to fragment the transmitted message to allow the transmission of large binary messages over a limited SMS message channel at the PUSH layer of WSP messages. On the SET side, the WSP protocol provides indexing for fragmented SMS message transmission. The first SULP message contains the size of the transmitted message.

An example of the first SMS message is shown in Figure 75. An example of subsequent SMS messages is shown in Figure 76.

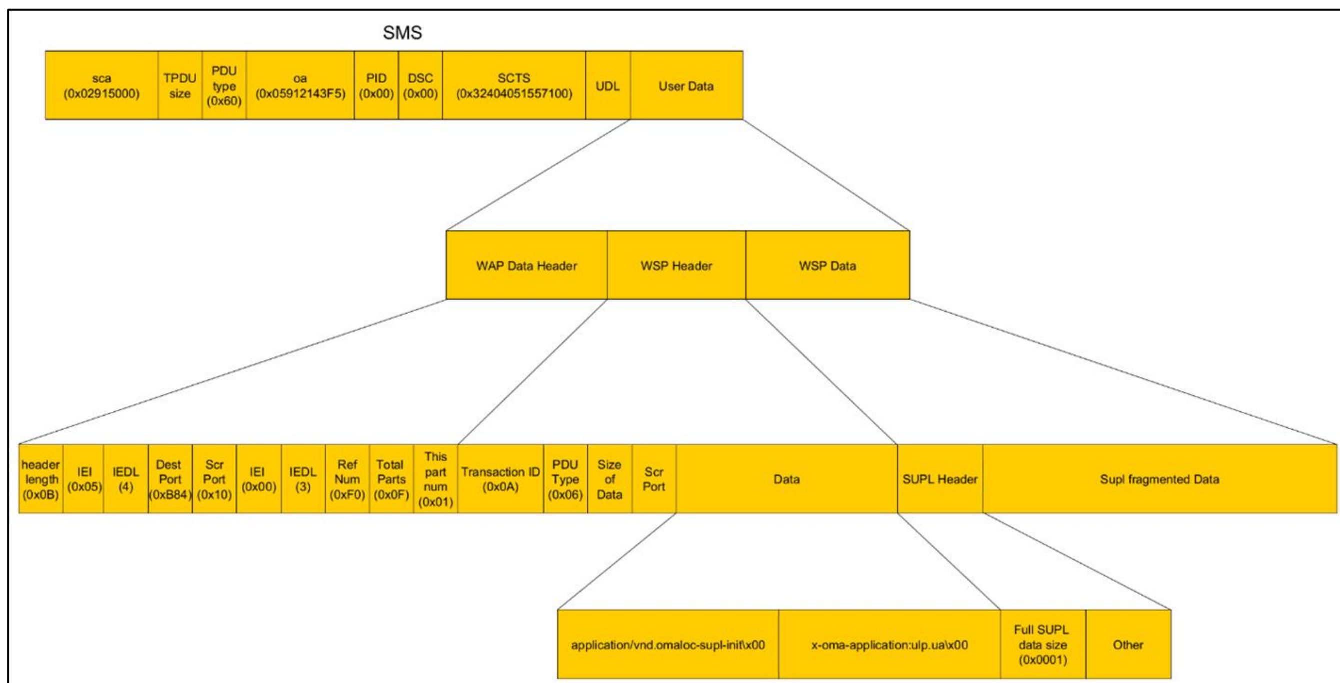


Figure 75. Example of the first SMS message

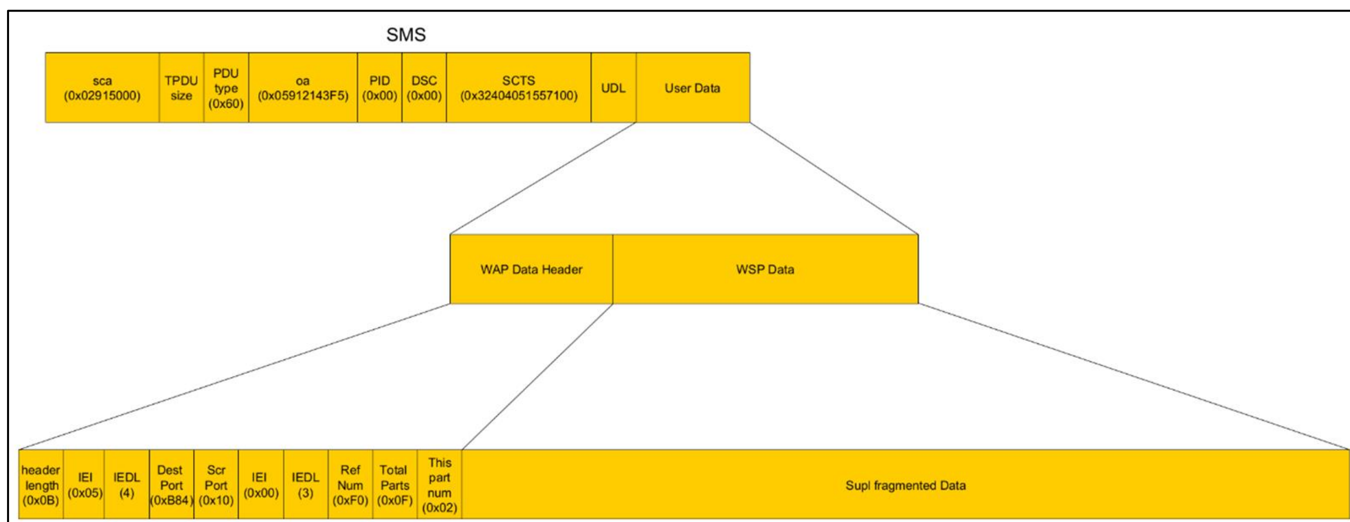


Figure 76. Example of subsequent SMS messages

The *ULPSizeFromPacket* variable is responsible for the size of the entire ULP packet, and *wapTpduLen* is responsible for the size of the received WAP message. WAP message processing consists of copying ULP message fragments into a buffer whose size is equal to *ULPSizeFromPacket*.

While analyzing the driver that is responsible for handling the fragmentation of ULP messages, we discovered a heap overflow vulnerability.

According to the transmission protocol, the *ULPSizeFromPacket* and *wapTpduLen* variables are calculated independently. These variables are interrelated only in the algorithm for receiving WAP messages: the sum of the sizes of all received WAP messages in a UPL message must not exceed *ULPSizeFromPacket*. But the algorithm used to receive WAP messages does not check this condition. Therefore, a received WAP packet of size

wapTpduLen will be copied to a buffer of size *ULPSizeFromPacket* (Figure 77). This is a classic heap-based buffer overflow.

```
    goto LABEL_44;
  }
  j_mem_fill_zero(v29, v43 + 1);
  j_memcopy((_BYTE *)Wap_Buffer_base, ULPSizeFromPacket, wapTpduLen);
  v30 = wapTpduLen;
}
```

Figure 77. Heap overflow

After generating an appropriate SMS message, we managed to cause a heap overflow error on the modem in one attempt, resulting in a hard reboot. To determine the cause of the reboot, we used the already known *AT+XLOG* AT command for reading error messages (Figure 78).

```
Date: 2018:1:1
 Time: 1:42:24
 Register:
 r0:  0xDDDDDDDD  r1:  0x00000132  r2:  0x60616CC0
 r3:  0x00000000  r4:  0x605C4BD8  r5:  0x00000008
 r6:  0x60616CC0  r7:  0x00000000  r8:  0xFFFF229C
 r9:  0xFFFFEEEE  r10:  0x605C4CD8  r11:  0xFFFF2C48
 r12:  0x60616CC0  r13:  0xFFFF3B20  r14:  0x98F184AD
 r15:  0x62BCB220
 SPSR: 0x200000D3  DFAR:  0xDDDDDDE1  DFSR: 0x00000005

OK
```

Figure 78. Reason for the reboot

The resulting dump made clear that the R0 register contained data that we controlled. Thus, we confirmed our ability to not only overflow the heap, but also embed our data into executable code.

However, we had no way to get a dump of memory at the moment of the crash. To understand whether the discovered vulnerability is serious or just another non-exploitable BoF, we had to solve the problem of reading the RAM.

7.8 How to read memory very, very slowly

After analysing the code at the address specified as the crash location in the AT command dump, we clearly understood how our data ended up in the R0 register (Figure 79). The crash consistently occurred in the *malloc()* function. The cause of the crash was an attempt to dereference an unmapped address, resulting in a memory access error and, consequently, a hardware fault.

```
do
{
  if ( Curr_Chunk[1] != 0xFFFFEEEE )
  {
    Curr_Chunk = (int *)*Curr_Chunk;        // Next chunk
    goto LABEL_22;
  }
```

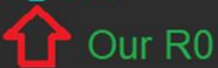


Figure 79. The reason for our data appearing in the R0 register

To understand why, we had to do a deep dive into the memory manager of the modem OS.

7.8.1 HeapFree and HeapBase structures

All heap memory is segmented into chunks. Each chunk contains a pointer to the next chunk, some user data (or unoccupied memory), and an indication of whether the chunk is currently occupied or free.

A chunk is considered free if its *heap_free_magic* value is *0xFFFFFFEEEE*. If the chunk is occupied, then instead of *heap_free_magic*, the field contains a pointer to the global structure that describes the current state of the whole heap (*HeapBase*) (Figure 80). The full list of fields in this structure can be found in the ThreadX operating system headers. These headers are publicly available[14].

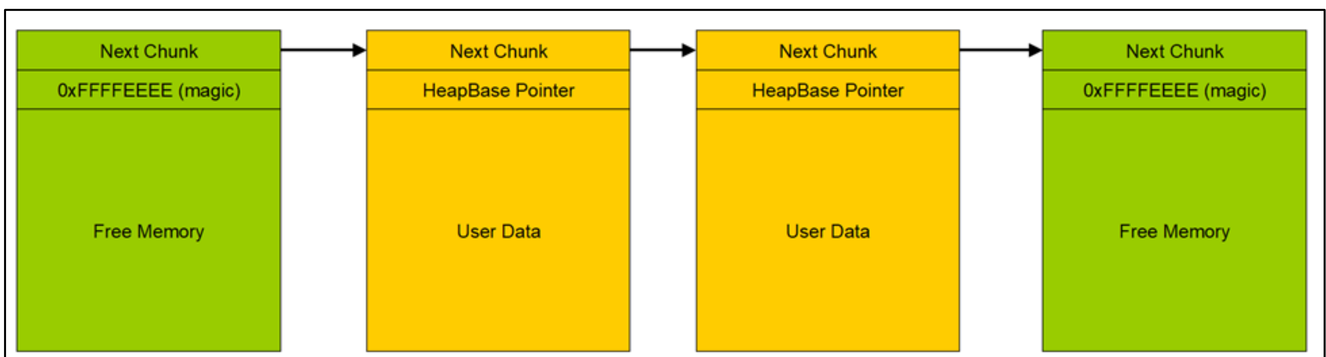


Figure 80. Heap structure

The search for a suitable free chunk is performed by traversing all free chunks, starting from the first one. The pointer to the first free chunk is stored in the *HeapBase* structure. The size of the current chunk is calculated by the difference of addresses between the next and the current chunk.

In the *malloc()* function, the search for a suitable free chunk involves a sequential traversal of a unidirectional linked list. If our SMS message exceeds the size of the allocated chunk, it overwrites the data in the next chunk when copied, corrupting it. However, this corruption did not immediately cause a software error or crash the modem. The OS continued to function, allowing the *malloc()* function to be called again, which involved traversing all chunks starting from the first free one.

Thus, while executing *malloc()*, there may be a moment when a free chunk is placed before the corrupted chunk. When the function attempts to move to the next chunk using the link

[14] https://github.com/eclipse-threadx/threadx/blob/0d308c7ae62085e68c7aa0a516f664c39e9a8407/common/inc/tx_api.h#L632C36-L632C36

from the corrupted chunk (Figure 81), the data in the R0 register is dereferenced to an address we control.

Figure 81. Executing malloc()

If the data is read from a valid address available in the modem's memory, no error occurs, and its value is stored in the R0 register.

```
    if ( Curr_Chunk[1] != 0xFFFFEEEE )
    {
      Curr_Chunk = (int *)*Curr_Chunk;          // Next chunk
      goto LABEL_22;
    }
    if ( !v9 )
    {
      v9 = 1;
      if ( (int *)CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_search == v13 )
        CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_search = (int)Curr_Chunk;
    }
    NextBlob = (int *)*Curr_Chunk;
    Curr_Chunk_size = *Curr_Chunk - (_DWORD)Curr_Chunk - 8;
    if ( Curr_Chunk_size >= REQ_MEM_SIZE )
      break;
    Curr_Chunk_size = 0;
    if ( NextBlob[1] == 0xFFFFEEEE )
    {
      *Curr_Chunk = *NextBlob;                  // next-> next
      CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_fragments = BASE[3] - 1;
      if ( (int *)CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_search == NextBlob )
        CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_search = (int)Curr_Chunk;
      goto LABEL_22;
    }
    Curr_Chunk = (int *)*NextBlob;
    if ( v14 )
    {
      --v14;
LABEL_22:
      if ( v14 )
        --v14;
    }
    __set_CPSR(v10);
    v10 = __get_CPSR();
    __disable_irq();
    if ( CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_owner != off_FFFF2C48 )
    {
      tx_byte_pool_fragments = CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_fragments;
      Curr_Chunk = (int *)CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_search;
      CONTAINING_RECORD(BASE, pool_ptr, field_0)->tx_byte_pool_owner = off_FFFF2C48;
      v14 = tx_byte_pool_fragments + 1;
    }
  }
  while ( v14 );
```

Decrement the total chunk counter

Figure 82. Reading data

However, a memory access error occurs if the address in R0 points to an unmapped memory location. In this case, the R0 register contains the last successfully read data. In our case, it is the data from the address we placed in the WAP message body.

7.8.2 Memory map

To understand which addresses can be considered valid, we needed to construct a map of the modem memory. To do this, we used the vendor-specific command to get information about memory regions: *AT@init:get_mem_info()*. The following table shows the resulting memory map.

Start region	End region	Description
0x00080000	0x000A0000	PSI RAM
0x62B80000	0x63E7FFFF	Code Section
0x60000000	0x62694C48	Ram Section
0xFFFF0000	0xFFFF1010	Mapped PSI RAM
0xFFFF2000	0xFFFF4000	Mapped PSI RAM
0x00400000	0x00500000	FLASH
0x18000000	0x19000000	HW
0xE0000000	0xF0000000	HW

Table 2. Memory map

Note that the handler of the above command reads memory regions from a table that is hardcoded in the firmware.

7.8.3 Reading memory via a heap overflow

Once we figured out which addresses were mapped, it became clear what kind of memory-read strategy we should use.

If we read data from the code section, the data would represent ARM machine instructions, and therefore, there will be unmapped addresses in memory. Thus, we can read from any address in the code section, including the BootROM (Figure 83).

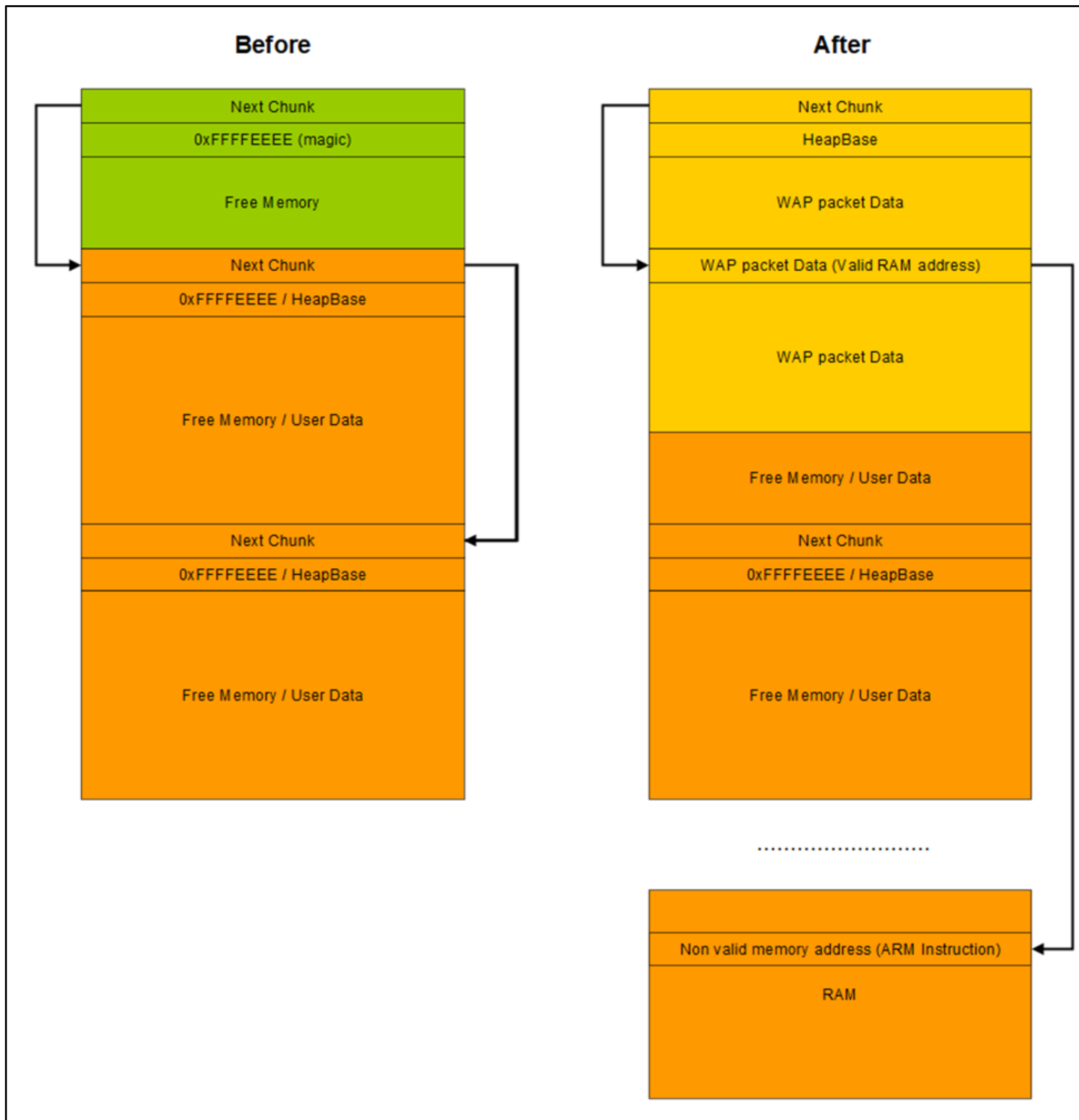


Figure 83. Reading a code section

Reading the data section required a slightly different approach (Figure 84). In the data section, unlike the code section, 4-byte words can potentially represent a mapped address in the modem's memory. In this case (and if the modem's CPU configuration allows it), unaligned addresses can be read instead of an address aligned by 4. If any cyclic

permutation of 4 bytes of read data is a mapped address, the real value stored at this address will not be read in the described manner. However, we have not found such addresses.

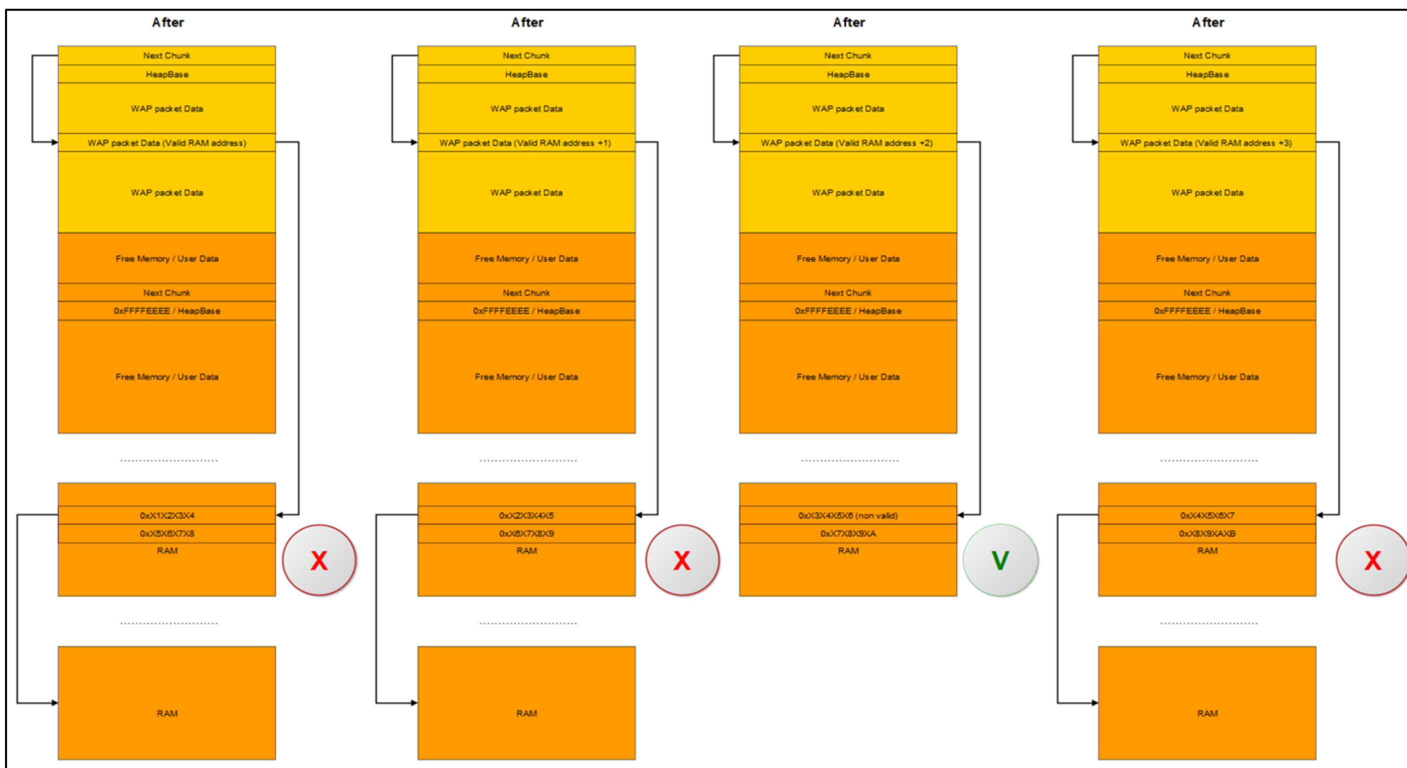


Figure 84. Reading the data section

As a result, we were able to read all the memory regions we were interested in. Most importantly, we discovered the environment context of the WAP SMS buffer overflow. It should be noted that the reading speed achieved using the described algorithm was only 7 b/min, so it took several weeks to read all the memory we wanted to.

7.8.4 Getting the overflow context

The method of reading memory described above assumes that the modem will reboot after each read of 4 bytes from the data section. However, because the system is loaded sequentially, the layout of the heap in the region where the overflow occurs never changes. Moreover, we found that the overflow itself always occurs at the same address!

This was possible because we achieved the overflow using the value of 1 for the *ULPSizeFromPacket* variable. This means that the very first free chunk suits us.

Due to the specifics of the OS boot process, the heap always has a small chunk of 8 bytes that is not suitable for any process (no *malloc()* calls request such a small amount of memory). Simultaneously, there are chunks allocated in memory that will not be freed until the OS shuts down, so this 8-byte chunk is never defragmented.

Every time we reboot the modem, we cause an overflow at the same memory address. Because we already know the address of the buffer overflow, we do not need a memory read primitive based on the discovered vulnerability.

7.9 Write primitive

Once the overflow context was established, we proceeded to the task of finding the write primitive. This primitive is often implemented by manipulating pointers to neighbouring chunks in linked lists with the accompanying injection of addresses into the global heap descriptor structure.

However, in our case the heap uses a single-linked list, and the only pointer we can control in the *HeapBase* structure is the pointer to the first free chunk in the heap.

Therefore, we needed to identify which data structures we could control that would let us write to memory, and to learn how to assemble these structures in memory.

We started to analyze the available functions affected by chunk overflow on the heap: the *malloc()* and *free()* functions. We needed to find a code fragment that would write to an address whose value could be affected by data from a WAP message.

7.9.1 Free function

The modem OS is a multitasking operating system: the heap memory pool is shared among many processes. To ensure that access to shared resources is consistent, the *HeapBase* structure stores a pointer to the process currently working with the heap. The *free()* function checks if the current process needs memory after the memory is freed (Figure 85).

```
if ( BASE_PTR[9] == v9 )
{
  v21 = *(_DWORD *)(v9 + 0x74);
  if ( v21 == v9 )
  {
    v22 = 0;
    BASE_PTR[9] = 0;
  }
  else
  {
    BASE_PTR[9] = v21;
    *(_DWORD *)(*(_DWORD *)(v9 + 0x74) + 0x78) = *(_DWORD *)(v9 + 0x78);
    *(_DWORD *)(*(_DWORD *)(v9 + 0x78) + 0x74) = *(_DWORD *)(v9 + 0x74);
    v22 = BASE_PTR[10] - 1;
  }
  BASE_PTR[10] = v22;
  *(_DWORD *)(v9 + 0x6C) = 0;
  ++dword_FFFF2C60;
    set_CPSR(CPSR);
  **(_DWORD **)(v9 + 0x80) = v20;
  *(_DWORD *)(v9 + 0x88) = 0;
  resume_suspend_thread((_DWORD *)v9);
  CPSR = __get_CPSR();
  __disable_irq();
}
else
{
  *(v20 - 1) = 0xFFFFEEEE;
  BASE_PTR[2] += *(v20 - 2) - (_DWORD)(v20 - 2);
  if ( BASE_PTR[5] > (unsigned int)(v20 - 2) )
    BASE_PTR[5] = v20 - 2;
}
```

If current thread is the pool owner?

Update Thread Structure

Figure 85. The free() function

This action is crucial for us, because it is the only place in the entire code of the *free()* and *malloc()* functions that writes to a memory address extracted through a double dereference. This is exactly what we were looking for in order to write to an arbitrary memory address!

A situation may arise where a process owns the heap memory pool and has a previously unprocessed memory allocation request. After returning the just freed memory to the pool of free memory, the *free()* function attempts to find a suitable free chunk of memory for allocation. Only if a suitable free chunk is found, the *free()* function update the *Thread* structure of the current process with information about the chunk.

This causes a pointer to the found free chunk to be written to the address specified in the *Thread* structure at offset *0x80*.

Instead of passing the original *Thread* structure of the current process to the *free()* function, we could provide a pointer to a crafted data structure that mimics the *Thread* structure. This makes it possible to write to any address in memory. Now we aren't writing some arbitrary piece of data, but a pointer to a free memory chunk. This was enough to intercept the flow of execution.

But the pointer to the *Thread* structure is taken from the *HeapBase* structure located at a static address in memory. The pointer to the *HeapBase* structure is contained in each occupied chunk at offset +4 and is used to access the *HeapBase* structure.

Consequently, if we overwrite that pointer with our data, we can completely replace the *HeapBase* structure that the *free()* function will work with, and, therefore, the *Thread* structure as well.

Next, we needed to understand the inner workings of these two structures and to find out which fields must be filled in (Figure 86) for the execution of the *free()* function to reach the code we were interested in.

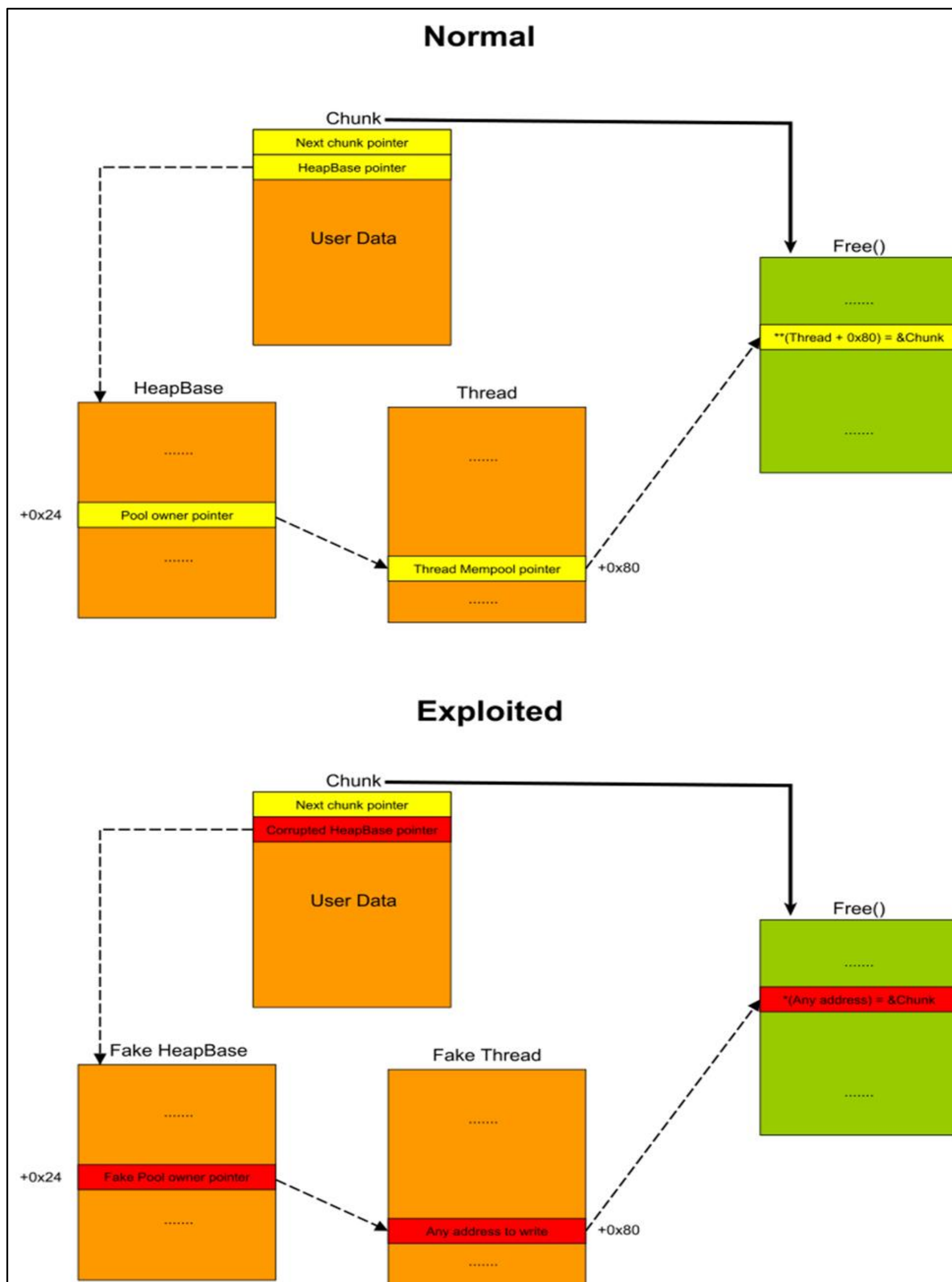


Figure 86. Our objective

7.9.2 Thread structure

The *free()* function uses only some of the fields of the *Thread* structure (Figure 87).

+0x6C	Any valid address
+0x70	Arbitrary
+0x74	Fake thread struct pointer
+0x78	Arbitrary
+0x7C	Memory size to allocate
+0x80	Address to write
+0x84	Arbitrary
+0x88	Any valid address

Figure 87. Fields of the Thread structure in the context of the free() function

The size of the chunk the process tries to allocate is at offset *0x7C*. This is the chunk that the *free()* function will try to find in the heap. We can use this field to control which chunk is assigned. This is important because our WAP SMS data is located on the heap. And if we choose a suitable memory allocation size in the free function, we will be able to write not only the address of a suitable free chunk to an arbitrary address, but the address of a chunk that contains our data!

The address where the pointer to the chunk is to be written to the heap if it is allocated successfully is located at offset *0x80*. Here we can have an arbitrary write address.

Data at offsets *0x74* and *0x78* can be rewritten during the execution of the *free()* function to values obtained by dereferencing current pointers. However, if there is a pointer to our *Thread* structure at offset *0x74* (verifying that the heap is still occupied by the same process), then nothing will be written to offsets *0x74* and *0x78*.

It is also important to note that a zero is written to offsets *0x6C* and *0x88* (Figure 88). These addresses also need to be valid, and writing to them must not cause the OS to fail.

kaspersky

```
if ( BASE_PTR[9] == v9 )
{
  v21 = *(_DWORD *)(v9 + 0x74);
  if ( v21 == v9 )
  {
    v22 = 0;
    BASE_PTR[9] = 0;
  }
  else
  {
    BASE_PTR[9] = v21;
    *(_DWORD *)(*(_DWORD *)(v9 + 0x74) + 0x78) = *(_DWORD *)(v9 + 0x78);
    *(_DWORD *)(*(_DWORD *)(v9 + 0x78) + 0x74) = *(_DWORD *)(v9 + 0x74);
    v22 = BASE_PTR[10] - 1;
  }
  BASE_PTR[10] = v22;
  *(_DWORD *)(v9 + 0x6C) = 0;
  ++dword_FFFF2C60;
    set_CPSR(CPSR);
  **(_DWORD **)(v9 + 0x80) = v20;
  *(_DWORD *)(v9 + 0x88) = 0;
  resume_suspend_thread((_DWORD *)v9);
  CPSR = __get_CPSR();
  __disable_irq();
}
else
{
  *(v20 - 1) = 0xFFFFEEEE;
  BASE_PTR[2] += *(v20 - 2) - (_DWORD)(v20 - 2);
  if ( BASE_PTR[5] > (unsigned int)(v20 - 2) )
    BASE_PTR[5] = v20 - 2;
}
```

Figure 88. Offsets 0x6C and 0x88 in the Thread structure

No other offsets in this structure are used during execution of the *free()* function. Therefore, we only need to imitate *a Thread* structure from offset *0x6C* to offset *0x88*.

7.9.3 *HeapBase* structure

After solving the question of how to fill in the fields of our simulated *Thread* structure, we needed to analyze the *HeapBase* structure. As shown in the table below, the *free()* function only uses a few fields of this structure.

Index in the *HeapBase* structure	Description
0x00	Magic of this structure: 'BYTE'
0x02	Size of available free memory in the heap now
0x05	Pointer to the next free chunk
0x09	Pointer to the beginning of the *Thread* structure (we are imitating only a part of the thread structure, so the pointer should be shifted accordingly).
0x0A	Number of requests to perform in the search for free chunks for OS processes. In our case, the value is 1.

Table 3. Fields of the HeapBase structure used in the free() function

The size of the available memory at offset *0x02* must be larger than the memory that the *free()* function will try to allocate for the simulated thread (offset *0x7C* of the *Thread* structure).

The pointer at offset *0x05* must not only point to a free chunk, but also be large enough to be allocated to our simulated thread (offset *0x7C* of the *Thread* structure).

No other offsets of this structure are used during execution of the *free()* function.

After preparing the structures, we had to develop an algorithm for forming these structures in the modem memory by sending WAP messages.

The process of receiving and processing WAP messages involves not only WAP protocol algorithms, but also the algorithm for processing free and occupied chunks as memory for messages is allocated and released. Further analysis of these algorithms helped us form the necessary *Thread* and *HeapBase* structures in the modem memory.

7.9.4 Strategy for handling free and occupied chunks

The *free()* and *malloc()* functions are called several times during the processing of incoming WAP SMS messages. Therefore, even if an overflow is achieved, we need to ensure that several conditions are met:

- No errors occur during the execution of the *free()* and *malloc()* functions.
- Our *Thread* and *HeapBase* structures are created and persist.

The memory manager's handling of free and occupied chunks directly influences whether these requirements are satisfied. We already knew the following:

- The search for a free chunk starts from the pointer to the first free chunk in the *HeapBase* structure (offset *0x05*).
- The size of the current chunk is calculated as the difference between the address of the next chunk and the current chunk.
- Only a chunk whose *heap_free_magic* is equal to *0xFFFFFEEEE* is considered free.

7.9.5 Selecting a new chunk

Next, we analyzed what happens when a new chunk is allocated by the *malloc()* function. During allocation, the user data is overwritten with zeros. The first chunk of a suitable size is always allocated.

If the size of the chunk is too large (exceeds the size of the user data by 20 bytes), it will be fragmented: another chunk will be created right after the end of the user data.

If the allocation process identifies a free chunk but it is smaller than the required size, and the next chunk is also free, then a defragmentation process takes place: the two free chunks are merged into one.

7.9.6 Releasing an occupied chunk

When releasing a chunk with an address less than the current address of the first free chunk in the *HeapBase* buffer (offset *0x05*), its address is updated as the new address of the first free chunk in the *HeapBase* buffer.

Therefore, the first free chunk in the heap is always allocated if the SUPL message size is always set to 1. If the function for releasing memory is called in time, then sending different SUPL messages will result in writing to the intermediate buffer located at the same address.

All that remained for us to do was to call *free()* and *malloc()* in the required sequence.

7.9.7 Processing fragmented SMSes

We noticed a condition where the algorithm for receiving WAP SMS messages writes data to the same memory addresses. If the actual sizes of the sent SMS messages are different, we can shape our structures so that we don't have to overwrite the previously received data.

However, the size of a single SMS message is not large enough to allow us to create the necessary structures. Therefore, we had to consider the possibility of using the WAP message fragmentation algorithm. This way we could deliver all parts of the required data structures.

Each WAP SMS contains a sequence number in its header. When the first WAP SMS message is received, a buffer of size *ULPSizeFromPacket* is allocated on the heap, into which all WAP message fragments are copied. However, the code does not check whether the sequence number of the current WAP message has already been used.

This way of processing allows the first message to be followed by the last one. Accordingly, it is possible to overflow the ULP message buffer with an arbitrarily large amount of data. In this case, we control the pointer inside the overflowed buffer by changing the offset of the next SMS message by sending the current SMS of the desired size: during copying, the pointer is moved by this amount. Each subsequent ULP message fragment is copied into memory immediately after the previous one.

Thus, we can try to form the required data structures by using three kinds of WAP messages:

Message type	Message properties
First WAP SMS message	The size of the entire ULP message is taken from it, and the *malloc()* function is called to allocate a buffer to store the entire SUPL message.
Any message that isn't the last one	The contained data (ULP message fragments) is simply copied after the data from the previous message (previous ULP message fragment) into the buffer allocated when the first message was processed.
Last message	The contained data is also copied after the data from the previous message into the buffer allocated when processing the first message. Then the collected ULP packet data is passed to its handler, and the *free()* function is called.

If we send a second message with the same index as the first fragmented WAP SMS message, the code will check if a buffer has already been allocated for the ULP message. Then, a call to the *free()* function is made to free it - a new ULP message has been sent, so there is no point in keeping the old one (Figure 89).

The previously allocated buffer is in the first free chunk, so it will be allocated again in the next call to *malloc()* as it has the lowest address among all free chunks. The *malloc()* function is called again for the first fragment of the just received ULP message. However, due to the inner workings of the memory allocator described above, the same buffer will be allocated! This will result in the same buffer being allocated for all the first WAP SMS messages.

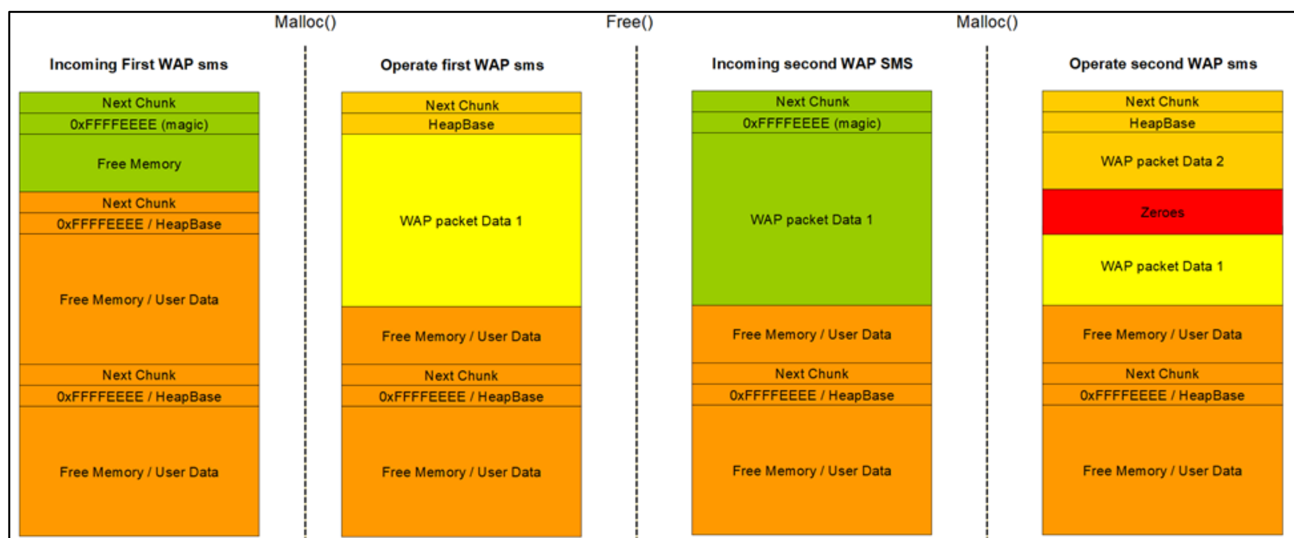


Figure 89. Specifics of sending fragmented WAP messages

The buffer for our first WAP SMS message is always allocated in the same memory location, so only the first 4 bytes of our ULP message will be zeroed out.

In a WAP SMS, we set the ULP message size to 1. Consequently, DWORD alignment means that the size of the buffer allocated in the *malloc()* function will be 4 bytes regardless of the size of the chunk that it will be placed in. And thanks to fragmentation, we can be sure that an overflow will overwrite the next chunk with our data. This is very important because in order for the entire chain of *free()* and *malloc()* calls to work, it must end with a call to *free()* on a chunk whose header data was overwritten by a pointer to our *HeapBase* structure. We can satisfy that requirement if we are able to control the header data of free chunks.

Using the previously described methods of writing to the heap, it was possible to not only store structures or code on the heap via WAP SMS messages, but also add a free chunk to the chain of all heap chunks. The very first WAP SMS message overwrites the pointer in the next chunk, which is already in the chain. We were able to overwrite this pointer with the correct address of the next chunk. The resulting crafted chunk resides inside our big chunk and will be the first to be found during the search for free chunks due to its location in the memory.

As a result, after creating the necessary data structures, free chunks and code in the modem's memory, we were able to start sending WAP messages again with the index corresponding to the first SMS in order to rewrite the data in the previously created free chunk. This ensured that the occupied chunk's *HeapBase* pointer was replaced with ours.

Now that we have created our own free chunk in the buffer, we need it to get claimed. And we must ensure this happens before we send a new SMS message that will overwrite that chunk of data. The algorithm for processing the received WAP message will help us achieve this (Figure 90).

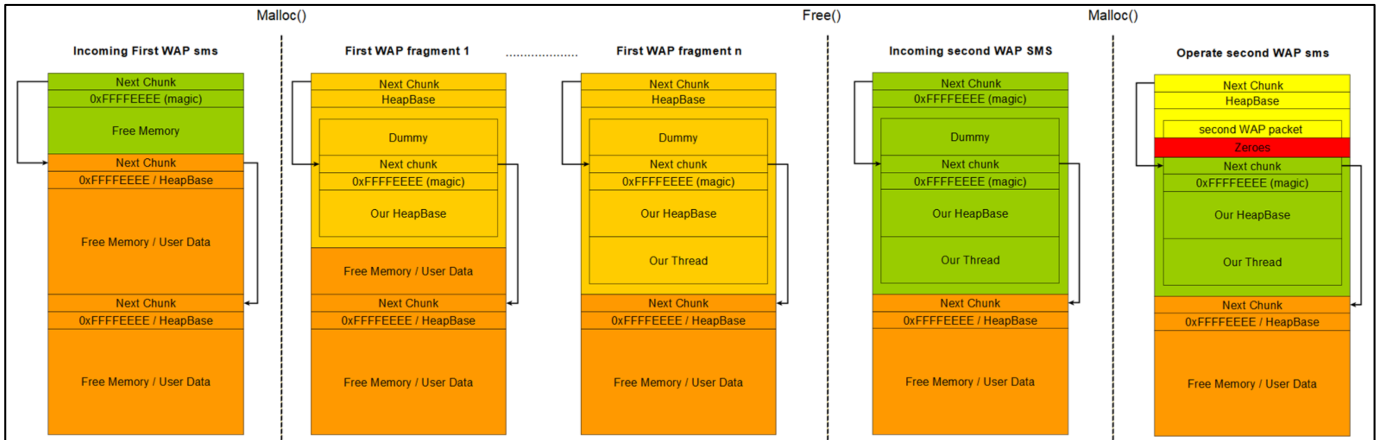


Figure 90. Algorithm for processing a received WAP message

7.9.8 Specifics of processing each WAP message

The message is copied to a temporary buffer allocated on the heap when each WAP message begins to be processed (Figure 91). From there, it is then copied to the buffer for the entire ULP message created during the processing of the first WAP SMS.

```
if ( DstPortNumber != 0x1C6B && DstPortNumber != 0xB84 )// check destination port number
{
  ((void (*)(const char *, ...))j_traceLog_0)(
    "Pos__Cat_GPSMngr_handleSUPLsms : SMS is not for LCS. Returning UTA_FALSE",
    v10,
    v8);
  return 0;
}
v40 = (char *)((int (__fastcall *)(_DWORD))j_cat_memalloc)(v44 + 1);
if ( !v40 )
{
  *(_DWORD *)&WAP_ref_number = -1;
  ((void (*)(const char *, ...))j_traceLog_0)(
    "Pos__Cat_GPSMngr_handleSUPLsms : Port number %d. Memory allocation failed. Returning UTA_FALSE",
    DstPortNumber);
  return 0;
}
j_memcopy(v40, (char *)v12, v44);
```

Figure 91. Copying a WAP message to a temporary buffer allocated on the heap

In this case, if the WAP message was not the last one, only the allocated temporary buffer is released after copying (Figure 92). The buffer containing part of the ULP message remains occupied.

```
if ( seqNumOfCurrentFrag < maxNumOfFragments )
{
  ((void (*)(const char *, ...))j_traceLog_0)(
    "seqNumOfCurrentFrag < maxNumOfFragments sending ACK for the current frag ment. waiting for next fragment. "
    "Returning UTA_TRUE");
  j_free_0((int)v40);      Free temp buffer
  return 1;
}
```

Figure 92. Releasing the allocated temporary buffer

In the figure above, you can see the *free()* function that we wanted to use. When receiving another WAP SMS message, we copy it to the free chunk we created inside the buffer. To do this, we sent a WAP SMS message with the data size equal to the size of the free chunk created inside our buffer. The *malloc()* function will start searching after this free chunk for

a suitable chunk to process the incoming WAP SMS. Thus, *malloc()* is guaranteed to return the pointer to our chunk!

All that remained to do was to make the current offset within the ULP message buffer point exactly to the beginning of the header of the newly allocated chunk.

7.9.9 Putting it all together

Thus, the final algorithm for exploiting the vulnerability consists of the following steps:

1. Send a WAP SMS message with the index of the first message.
2. By overflowing the ULP message buffer we overwrite the pointers and data of the next chunk and create free chunks whose size is *0x10* bytes inside our SUPL message buffer. It is important that there is an occupied chunk after the free chunk, otherwise defragmentation will happen!
3. Send the next WAP SMS message fragment. This message contains the *HeapBase* and *Thread* structures.
4. Send another WAP SMS message with the index of the first message. This time the message size should be large enough so that the next SMS fragment overwrites the beginning of our free chunk.
5. Send another SMS fragment (but not the one with the last index!). The size of the message must be of 8 bytes. These 8 bytes will overwrite the pointer to the *HeapBase* structure in our chunk.
6. When the WAP message processing function exits, the *free()* function is called and uses our *HeapBase* and *Thread* structures.

This procedure makes it possible to execute arbitrary code on the modem in just 4 SMS messages.

kaspersky

8 Post exploitation of network level vulnerabilities

8.1 Executing AT commands

We were finally able to unlock vendor-specific AT commands after we managed to write an arbitrary modem memory address as the value of our chunk pointer. To do this, we had to change our privilege level, which is located at the static address *0x600D0AD0*. Verification of the current privilege level is done by overlaying a binary mask that corresponds to the level. Therefore, to elevate our privileges in the modem's AT console, we had to write some large number at the memory address *0x600D0AD0* (Figure 93).

Figure 93. Privilege escalation

We can run the "*at@help*" command to ensure that all vendor-specific AT commands are unlocked. We see the full output of the help command only if the commands are unlocked (Figure 94).

```
at@help
at@help

Syntax: at@[24844<CRC>][&][<channel id>][,<transaction id>][<#>/<##>][<interfa
ce tag>:]<command sequence>
```

Figure 94. Vendor AT commands syntax

We can now get a list of all available vendor-specific AT commands along with their descriptions (Figure 95).

```
at@*:?
gticom        Common Test Control Interface v.0.0.2
xl1           XL1 trace interface v. 1.00.00
unf           UMTS RF v. 1.00.00
utif          UMTS test interface v.1.00.00
getif         GSM EDGE test interface v.1.00.00
gcal          2G RF driver test and calib. interface v.1.00.00
ucal          UMTS calibration interface v.1.10.00
fspeed        full speed test interface v.1.00.00
nvm           NVM interface v.0.01.00
prodif        Production Interface (prodif) v.2.00.00
prodctri      Production Control Interface (prodctrl) v. 1.00.00
driver        Ver test interface v. 01.00.0
pmu           PMU API AT test interface v.00.00.0
pow           POW API AT test interface v.00.00.0
pcl           pcl interface v.1.00.00
ts            time services test interface v.01.00.0
trap          trap debug interface v.1.00.00
meas          meas debug interface v.1.00.00
utasensor     UTA SENSOR interface v. 1.00.00
utabm         UTA BM debug interface v.1.00.01
init          Init test interface v.01.00.0
```

Figure 95. List of all available vendor AT commands

And finally, we read the modem memory without registering in our GSM network and without using any special ULP SMS (Figure 96).

```
at@xl1:mem_rdb(0x632C8518, 0x10)
at@xl1:mem_rdb(0x632C8518, 0x10)

632C8518: 78 60 01 98

632C851C: 86 42 0B D2

632C8520: 63 48 5F F0

632C8524: 40 EA 00 E0
```

Figure 96. Reading the modem memory

As a result, we have a full-fledged console for reading and writing data to the modem RAM, which we used to unlock the vendor-specific AT command *sec* without knowing the special manufacturer's key, switch the modem to manufacturer mode, and more.

8.2 Code execution?

Of course, we haven't yet executed any of our code. We already had the ability to write a pointer to our data at an arbitrary address in the modem's memory. This data could be binary code compiled for ARM. It could be sent to the modem via a WAP SMS. The final step was to find a suitable location to write the address of our code for intercepting the flow of execution.

We had the ability to write only a single DWORD, so we searched for a place in the modem OS code that would immediately transfer control to an address taken from RAM. We found such a place in the code of the OS process manager.

The process manager works with the *Thread* structure. We found that a function pointer could be stored at offset *0x94* in this structure. Pointers to specific OS kernel callbacks are often stored in this way.

Accordingly, we only needed to write a pointer to our ARM code in the heap memory at offset *0x94* of a thread descriptor (Figure 97).

To do this, we rewrote the *Thread* structure that belonged to the ULP message processor.

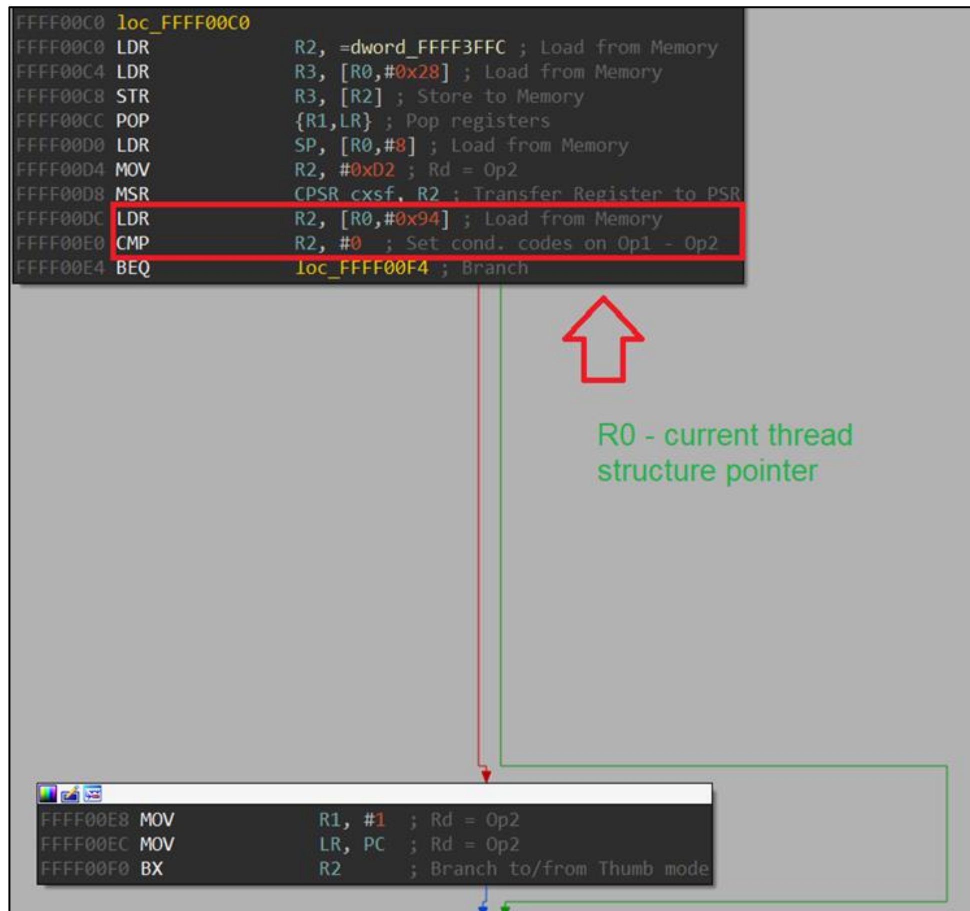


Figure 97. Preparing for code execution

As a result, we executed our code on the modem OS in the process manager's context. In doing so, we also confirmed that the data section is executable.

8.3 Configuring the MMU

The code fragment where control was intercepted happened to be a part of a critical section. In this code fragment, all interrupts are disabled, making it unviable for gaining persistence in the OS. Code in this mode must be executed as fast as possible to avoid triggering the watchdog, and no operations involving external components can be performed in this mode.

Therefore, we decided to use the code executed in the context of the process manager to modify the code of an OS thread. But the code section was mapped in *RX* (Read / Execute) mode. Thus, before taking any further action, it was necessary to make sure that the code section could be written to.

During the research, we determined that the kernel stores a complete MMU translation table at the logical address *0x00088000*. The code section and the data section were mapped in page mode, where the size of one page is *0x100000* bytes (1 MB). A 1:1 translation was used for these sections. This can be observed in the memory area where the MMU translation table is stored (Figure 98).

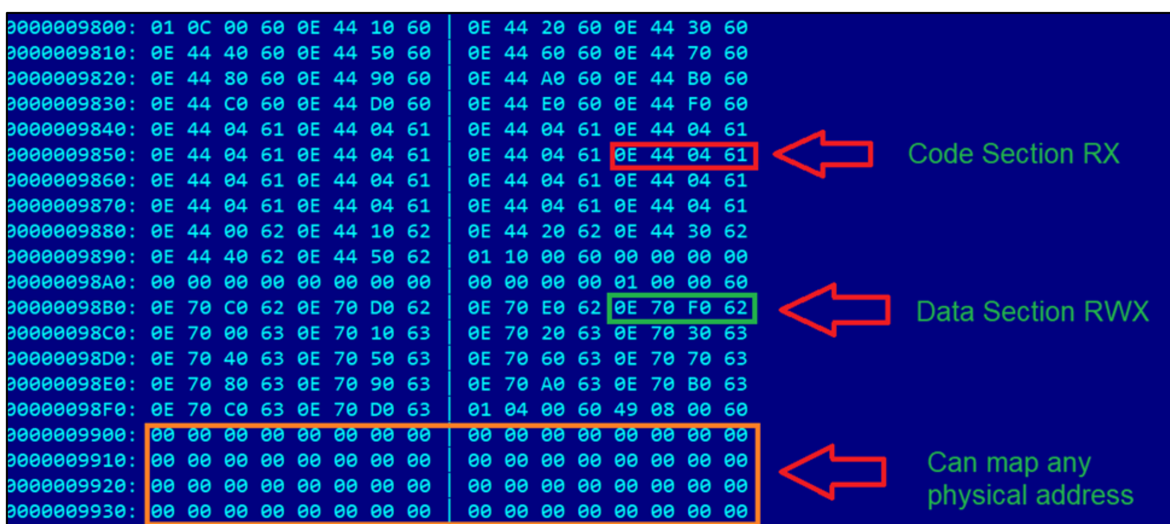


Figure 98. MMU settings

The section access mode setting was not standard one for the ARM architecture. Therefore, we decided to simply copy the access setting from the data section, which was mapped in RWX (read / write / execute) mode, as we have already seen. After that, we were able to write to the code section.

Additionally, we could now obtain a physical memory map of the modem. At offset *0x00089900* there were free logical addresses guaranteed to be unmapped. However, we could map them manually! This approach allowed us to supplement the already available information about the modem memory map with information about the exact size of available RAM.

8.4 SMS FS and OTAP activation

After unlocking the code section for writing, we needed to select a thread suitable for modification. *UTACAT* was chosen in order to then develop a half-duplex communication channel with the modem. This process is responsible for handling SMS messages. Its code was modified in the SMS processing function (Figure 99).

kaspersky

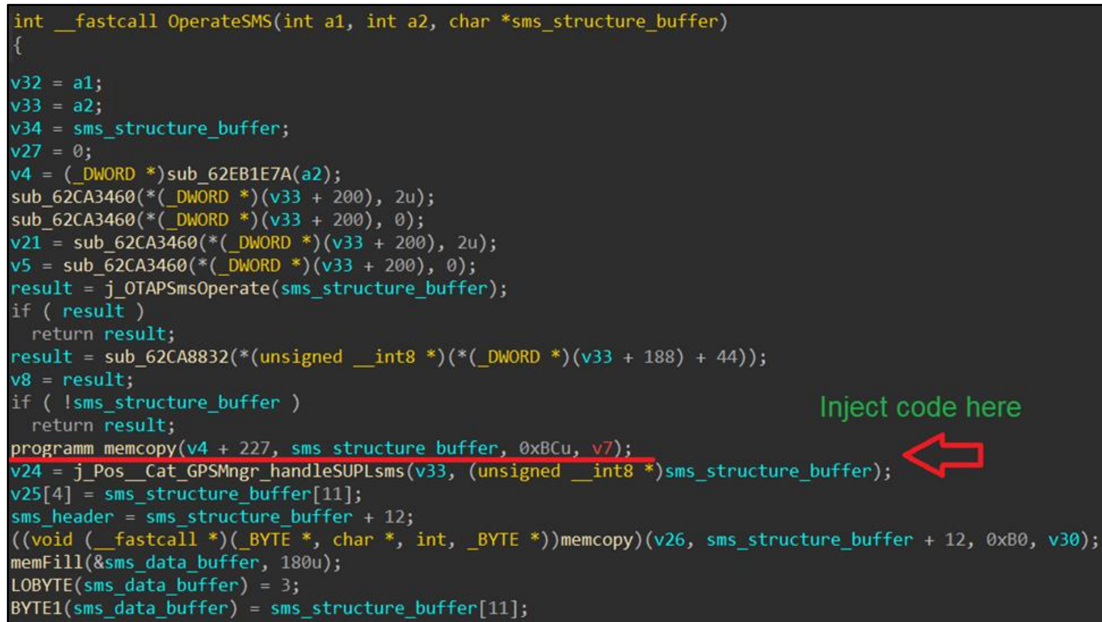


Figure 99. Modification of UTACAT

After determining the location that needed to be modified, we compiled a set of functions to work with the modem's UFS. These functions let us activate OTAP and then install our MIDlet.

Also, since most UFS functions work with local buffers, we needed additional memory handling functions to manage modem memory. The final list of functions included:

- Memory allocation function (*malloc*)
- Memory release function (*free*)
- Function for opening/creating a file in the UFS (*createFile*).

For each of the three functions, we needed to determine whether they could be invoked during SMS message handling and to identify their input parameters and their format.

Memory can be managed by the *malloc()* and *free()* functions. However, high-level functions for working with the modem's UFS were only found in the *JVM* process. Analysis of the modem OS code did not indicate the presence of memory sharing between processes, so we decided to use the functions of the *JVM* process within the *UTACAT* process.

The *malloc()* and *free()* functions have no special execution context. *malloc()* takes only one parameter as input – the size of the buffer to be allocated. The *free()* function accepts only a pointer to the buffer to be freed. They are shown in Figure 100.

```
int __fastcall j_cat_memalloc(int mem_size)
{
  return cat_memalloc_ext(mem_size);
}

int __fastcall j_free_0(int HeapBufferPointer)
{
  return free_0(HeapBufferPointer);
}
```

Figure 100. Parameters of the free() and malloc() functions

The remaining function for working with the UFS needed to provide the ability to create or overwrite an OTAP activation file. This function in the JVM process is used to work with the UFS. It takes only two parameters as input:

- absolute path to the file in the UFS

- operation mode

We found that '*wb*' mode corresponds to the value *0x6C* (Figure 101).

```
int __fastcall UTA_OpenFile_func(int FileFullPath, int OpenType)
{
  int v2; // r3
  int v5; // r4
  int v6; // r0
  int FSFilePath; // r0
  int v9[2]; // [sp+4h] [bp-24h] BYREF
  int v10; // [sp+Ch] [bp-1Ch] BYREF
  int v11; // [sp+10h] [bp-18h] BYREF

  if ( !FileFullPath )
  {
    v5 = -5;
    if ( sub_62C10CA0(0, 0xECu) )
    {
      v6 = 391;
LABEL_9:
      v9[0] = v6;
      v9[1] = v5;
      sub_62C11420(0xECu, 3, 9900, (int)v9, 8);
      return v5;
    }
    return v5;
  }
  v5 = sub_62C65FE4(FileFullPath, &v11, (int)&v10, v2);
  if ( v5 )
  {
    if ( sub_62C10CA0(0, 0xECu) )
    {
      v6 = 401;
      goto LABEL_9;
    }
    return v5;
  }
  FSFilePath = getFSPath(FileFullPath);
  return (*(int (__fastcall **)(int, int, int))(v11 + 0x3C))(FSFilePath, OpenType, v10);// sub_62C64B64
}
```

Figure 101. Function for creating/opening a file in the UFS

Following this research, we developed a small driver written in ARM assembly. The driver allows execution of all described functions via SMS messages (Figure 102).

```
_create_file_func:
LDR R5, [R1] //size to write
ADDS R1, #4
MOV R0, R1 //file name address
MOV R1, #0x6D //create file
LDR R2, =0x62C46CC9 //create or open file
BLX R2

_malloc_func:
LDR R0, [R1]
LDR R1, =0x62C9FECD //malloc()
BLX R1

_free_func:
LDR R0, [R1]
LDR R1, =0x62CA05E5 //free()
BLX R1
```

Figure 102. Driver for invoking functions via SMS messages

We decided to use the special magic value *0x6AA677BB* in the header to distinguish our SMS messages from ULP, OTAP or plain text messages (Figure 103).

```
LDR R2, =0x6AA677BB// our magic
SUB R4, R2, R4
CMP R4, #0
BNE exit_code
ADDS R1, #4
LDR R2, [R1]
CMP R2, #1
BEQ _malloc_func
CMP R2, #2
BEQ _free_func
CMP R2, #3
BEQ _create_file_func
```

Figure 103. Using the custom magic value 0x6AA677BB in the message header

By applying the described technique to execute our code, we were able to successfully run the driver within the context of the UTACAT process. This confirmed that the processes' RAM was not isolated from each other in any way: all data and code from one process is available to any other process. All we had to do was to create the necessary OTAP files on the FS and install our MIDlet (Figure 104).

```
[OTAP] Midlets stopped
[OTAP] PS detach success
[OTAP] Starting installation
[OTAP] Try to get http://            /helloworld.jad ...
[OTAP] Transfer finished.
[OTAP] JAR file download
[OTAP] Try to get http://            /helloworld.jar ...
[OTAP] Transfer finished.
[OTAP] Installation completed
```

Figure 104. Installing our own MIDlet

9 CVE list

CVE ID	CVSS Score	Description
CVE-2023-47610	8.1 (High)	A CWE-120: Buffer Copy without Checking Size of Input vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow a remote unauthenticated attacker to execute arbitrary code on the targeted system by sending a specially crafted SMS message.
CVE-2023-47611	7.8 (High)	A CWE-269: Improper Privilege Management vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow a local, low privileged attacker to elevate privileges to "manufacturer" level on the targeted system.
CVE-2023-47612	6.8 (Medium)	A CWE-552: Files or Directories Accessible to External Parties vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow an attacker with physical access to the target system to obtain a read/write access to any files and directories on the targeted system, including hidden files and directories.
CVE-2023-47613	4.4 (Medium)	A CWE-23: Relative Path Traversal vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow a local, low privileged attacker to escape from virtual directories and get read/write access to protected files on the targeted system.
CVE-2023-47614	3.3 (Low)	A CWE-200: Exposure of Sensitive Information to an Unauthorized Actor vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow a local, low privileged attacker to disclose hidden virtual paths and file names on the targeted system.
CVE-2023-47615	3.3 (Low)	A CWE-526: Exposure of Sensitive Information Through Environmental Variables vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow a local, low privileged attacker to get access to a sensitive data on the targeted system.
CVE-2023-47616	2.4 (Low)	A CWE-200: Exposure of Sensitive Information to an Unauthorized Actor vulnerability exists in Telit Cinterion BGS5, Telit Cinterion EHS5/6/8, Telit Cinterion PDS5/6/8, Telit Cinterion ELS61/81, Telit Cinterion PLS62 that could allow an attacker with physical access to the target system to get access to a sensitive data on the targeted system.

10 Conclusion

Though being a special-purpose device, a modern modem implements numerous features and potential user scenarios. In fact, it is a complicated system, both from an architecture and implementation point of view. Due to performance requirements, most of the key features are implemented in low-level languages such as C and Assembler and therefore lack built-in safeguards mitigating potential developers' mistakes.

In the course of the modem security analysis, we found seven locally exploited vulnerabilities and one remotely exploited vulnerability. The combination of these vulnerabilities could allow an attacker to completely get control over the modem. In our truck's security audit project, having control of the modem we were able to get our foothold in the telecommunication unit embedding it, and further, to propagate to other truck ECUs ending with getting control over the main vehicle systems, such as the engine, the gearbox, the suspension, the breaks, etc., therefore being able to totally compromise the vehicle safety from remote.

All discovered vulnerabilities have been reported to the vendor. Some of them have not been addressed by the vendor so far as the product support discontinued. And even if the vendor fixed all the vulnerabilities, as we stated at the beginning of the report, in some cases, the modem is integrated in such a way that applying updates would be difficult.

Thus, to counter the threats posed by the found vulnerabilities, Kaspersky recommends:

- Contact the mobile operator to disable the sending of SMS messages to the device.
- Use private APN with carefully configured security settings to limit the impact of any potential exploit.
- Enforce application signature verification to prohibit the installation of untrusted MIDlets on the device.
- Control physical access to the device at all stages of transportation to protect against the embedding of backdoors.
- When developing a new product consider remote modem compromise as a high potential risk and restrict accordingly access from the modem (or the unit embedding it) to other products' mission-critical components.

As for the vendors of the modems and similar devices, to mitigate potential risks at the design stage, Kaspersky recommends:

- Introduce additional memory access restrictions in the ThreadX operating system.
- Use static code analysis tools to determine if there are any errors in logic or pointer arithmetic.
- Perform fuzz testing ("fuzzing") for the application to find implementation bugs using malformed/semi-malformed data injection in an automated fashion.
- Perform code walk-through audits to look for confusing logic and other errors.
- Select the development tool stack enforcing security domain separation and promoting a Secure by Design approach such as the one advocated by the KasperskyOS developers.