Kaspersky ICS CERT

kaspersky

# Динамический анализ компонентов прошивок IoT-устройств

Сергей Ануфриенко

06.07.2022 Version 1.0

Анализ прошивки — необходимая составляющая исследования безопасности и направленного поиска уязвимостей продуктов IoT, компонентов автомобилей, систем АСУ ТП и множества программно-аппаратных комплексов прочих типов и назначений.

Общий размер прошивки различных устройств и объём программного кода в отдельных бинарных файлах часто довольно большой. Поэтому в таких случаях при исследовании прошивок целесообразно с точки зрения экономии времени и сил попытаться произвести динамический анализ — посмотреть, как работает тот или иной код, найти цепочку вызовов, приводящую к запуску той или иной ветки, провести фаззинг и т. д.

В статье мы рассмотрим как традиционные способы динамического анализа —

- связку QEMU и GDB и отладку непосредственно на целевой системе,

так и менее очевидные, но тем и интересные, способы:

- эмулятор Renode — инструмент для полносистемной эмуляции, которому до сих пор сообщество исследователей безопасности уделяло незаслуженно мало внимания;
- фреймворк Qiling — инструмент для эмуляции API ОС и окружений (таких как UEFI), схожий по своей природе с `qemu-user`, но имеющий большую гибкость и приспосабливаемость, так как написан на языке высокого уровня Python.

Каждый инструмент имеет свои сильные и слабые стороны и в различной степени подходит для решения определенного круга задач.

Мы продемонстрируем некоторые возможности этих инструментов на примере прошивки сетевого видеорегистратора одного из крупных производителей — при этом исследование будет проводиться в условиях отсутствия в нашем распоряжении самого устройства.

А в качестве бонуса рассмотрим отладку с помощью GDB непосредственно на другом устройстве, имеющемся в нашем распоряжении, — головном устройстве (head unit) от автомобиля известной марки.

## Алгоритм и объект исследования

На примере исследования прошивки сетевого видеорегистратора покажем некоторые трудности, с которыми сталкиваются исследователи безопасности при анализе прошивок устройств, и возможные пути их решения с использованием современного эффективного инструментария.

В целом алгоритм работы исследователя прошивок можно разделить на несколько этапов:

1. Определить формат прошивки, распаковать, проанализировать составные части многокомпонентных прошивок.
2. Выполнить первоначальный анализ полученных данных — определить целевую архитектуру, ОС, назначение отдельных файлов.
3. Провести статический анализ представляющих интерес частей прошивки, определить необходимость и объем динамического анализа.
4. Выбрать цели для динамического анализа и инструменты для его реализации.
5. Попытаться запустить исследуемый компонент на выполнение с использованием выбранного инструмента.
6. Расставить «заглушки» для обхода не нуждающихся в эмуляции и/или не поддающихся эмуляции частей программы, а также для задания начального состояния.
7. Проанализировать результаты работы исследуемого кода с помощью выбранного инструмента.

Выбранный для исследования прошивки видеорегистратор построен на платформе компании HiSilicon, а в качестве операционной системы в нем используется Linux. Скачанная с сайта производителя прошивка состоит из единственного файла, в котором в результате анализа утилитой `binwalk` обнаруживается образ файловой системы CramFS. После его распаковки внутри находим комбинированный образ ядра Linux и initramfs — `uImage`, а также несколько зашифрованных скриптов и `tar`-архивов.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CR
C: 0xCA9A1902, created: 2019-08-23 07:16:16, image size: 4414954 bytes, Data
Address: 0x40008000, Entry Point: 0x40008000, data CRC: 0xDE0F30AC, OS: Linux
, CPU: ARM, image type: OS Kernel Image, compression type: none, image name:
"Linux-3.18.20"
64            0x40            Linux kernel ARM boot executable zImage (little
-endian)
2464          0x9A0           device tree image (dtb)
16560         0x40B0          LZMA compressed data, properties: 0x5D, diction
ary size: 33554432 bytes, uncompressed size: -1 bytes
4401848       0x432AB8        device tree image (dtb)
```

После того, как базовая информация об объекте исследования получена, попробуем воспользоваться упомянутыми выше инструментами, чтобы получить больше информации, в том числе файлов, и определить среди них перспективные цели для дальнейшего анализа.

# Эмуляция на системном уровне с помощью Renode

Renode — это инструмент, позволяющий эмулировать целевую систему полностью, включая взаимодействие между собой нескольких виртуальных процессоров, каждый из которых может иметь свою архитектуру и прошивку, виртуальные микросхемы памяти, сенсоры, дисплеи и другую периферию. Кроме того, Renode позволяет связать между собой эмулируемое оборудование и реальное «железо», реализованное в микросхеме ПЛИС.

Стоит отметить, что Renode нацелен в первую очередь на эмуляцию Embedded/IoT-устройств, работающих под управлением встраиваемых ОС, хотя запуск полноценных ОС, таких как Linux или QNX, также возможен, и в репозитории проекта на GitHub есть соответствующие примеры. Список отладочных плат, для которых в Renode «из коробки» реализована поддержка по крайней мере некоторой периферии, можно посмотреть в документации.

В первую очередь Renode позиционируется разработчиками как инструмент, предназначенный для облегчения процесса разработки, отладки и автоматизированного тестирования встроенного ПО. Однако его можно использовать и для целей динамического анализа поведения исследуемых систем. Полезные возможности, которые предоставляет Renode для исследователя, включают:

- Поддержку архитектур ARM Cortex-A/M, x86, RISC-V, SPARC, POWER;
- Возможность эмуляции любой периферии на языке C#;
- Соединение виртуальных устройств между собой с использованием I2C, SPI, USB, Ethernet и других интерфейсов;
- Возможность подключения к эмулируемой системе отладчика GDB для отладки, а также анализа и изменения регистров CPU, состояния системной памяти, виртуальных устройств и т. д. в любой момент, в том числе программным путем;
- Возможность написания обработчиков определенных событий (например, чтение/запись определенных адресов памяти) на Python или C#;
- Возможность создания плагинов на .NET-языках, реализующих новые команды, например для импорта имен функций и переменных из IDA/Ghidra SRE.

## Описание эмулируемой платформы

Как правило, периферийные устройства, входящие в состав однокристальных систем, доступны посредством Memory Mapped I/O (MMIO) — регионов физической памяти, на которые отражаются регистры соответствующих периферийных модулей. Renode предоставляет возможность собрать систему на чипе как конструктор — из блоков, с помощью конфигурационного файла с расширением `.repl` (**RE**node **PL**atform), описывающего то, какие устройства следует разместить по каким адресам в памяти.

Информацию о доступных периферийных устройствах, а также карте распределения памяти в применяемой платформе можно почерпнуть из документации на SoC (при ее наличии в открытом доступе), а при ее отсутствии, например, проанализировав содержимое DTB (Device Tree Blob), — блока данных с описанием платформы для ядра Linux, который необходим для запуска Linux на встраиваемых устройствах.

В исследуемой прошивке блок DTB, по информации от того же `binwalk`, присоединен в конец файла `uImage`. Преобразовав DTB в читаемый формат (DTS) с помощью утилиты `dtc`, мы можем на его основе составить описание платформы для Renode:

```
uart0: UART.PL011 @ sysbus 0x12080000
    -> gic@6
    size: 0x1000

uart1: UART.PL011 @ sysbus 0x12090000
    -> gic@7
    size: 0x1000

uart2: UART.PL011 @ sysbus 0x120a0000
    -> gic@8
    size: 0x1000

uart3: UART.PL011 @ sysbus 0x12130000
    -> gic@20
    size: 0x1000

timer0: Timers.SP804 @ sysbus 0x12000000
    -> gic@1

memory: Memory.MappedMemory @ sysbus 0x40000000
    size: 0x8000000

sysCtl: Miscellaneous.ArmSysCtl @ sysbus <0x12050000, +0x1000>
    procId: 0x0C000191

pl310: Cache.PL310 @ sysbus <0x10700000, +0x10000>

gic: IRQControllers.GIC @ {
        sysbus new Bus.BusMultiRegistration { address: 0x10301000; size: 0x1000; reg
ion: "distributor"};
```

```
        sysbus new Bus.BusMultiRegistration { address: 0x10300100; size: 0x100; regi
on: "cpuInterface"}
    }
    0 -> cpu@0
    itLinesNumber: 2
    numberOfCPUs: 1

cpu: CPU.Arm @ sysbus
    cpuType: "cortex-a9"

sysbus:
    init:
        Tag <0x12080000, 0x12080FFF> "UART0"
        Tag <0x12090000, 0x12090FFF> "UART1"
        Tag <0x120A0000, 0x120A0FFF> "UART2"
```

В описание платформы для Renode был взят минимальный набор из представленных в Device Tree периферийных устройств: последовательные интерфейсы (UART), системный таймер, оперативная память, контроллер прерываний. Этого должно быть достаточно, чтобы хоть что-то запустилось, — а дальше можно действовать по ситуации. Для примера, так выглядит описание одного последовательного интерфейса в DTS:

```
uart@12080000 {
    compatible = "arm,pl011\0arm,primecell";
    reg = <0x12080000 0x1000>;
    interrupts = <0x00 0x06 0x04>;
    clocks = <0x02 0x23>;
    clock-names = "apb_pclk";
    status = "okay";
};
```

В данном случае нам повезло, и все нужные устройства уже были доступны в библиотеке Renode — последовательный порт PL011, таймер SP804, стандартный контроллер прерываний ARM. Однако часто бывает необходимо написать минимальную реализацию для какого-то устройства, которого нет в составе Renode, либо использовать теги-заглушки (об этом ниже).

## Запуск эмуляции

Для того чтобы запустить что-то полезное на полученной платформе, необходимо подготовить скрипт инициализации. Как правило в таком скрипте происходит загрузка исполняемого кода в виртуальную память, настройка регистров процессора, установка дополнительных обработчиков событий, настройка вывода отладочных сообщений (если необходимо) и др.

```
:name: HiSilicon
:description: To run Linux on HiSilicon

using sysbus
$name?="HiSilicon"
mach create $name
machine LoadPlatformDescription @platforms/cpus/hisilicon.repl
logLevel 0
sysbus LogAllPeripheralsAccess true

### create externals ###
showAnalyzer sysbus.uart0

### redirect memory for Linux ###
sysbus Redirect 0xC0000000 0x40000000 0x8000000

### load binaries ###
sysbus LoadBinary "/home/research/out/uImage" 0x40008000
sysbus LoadAtags "console=ttyS0,115200 mem=128M@0x40000000 nosmp maxcpus=0" 0
x8000000 0x40000100


### set registers ###
cpu SetRegisterUnsafe 2 0x40000100 # atags
cpu PC 0x40008040
```

Скрипт загружает `uImage`-файл в память платформы по адресу, который был получен из вывода `binwalk`, настраивает аргументы ядра (Linux ожидает, что `ATAGS` передаются по смещению `0x100` от начала оперативной памяти, также этот адрес передается в регистре `r2`) и передает управление по адресу `0x40008040`, так как первые `0x40` байт — это заголовок `uImage`.

В скрипте инициализации также можно выполнить множество дополнительных действий: установить другие регистры процессора (например, если выполнение кода будет начинаться не с точки входа); записать произвольное значение в память платформы (например, если нужно модифицировать какие-то инструкции в загруженном на предыдущих шагах исполняемом коде); включить сервер GDB или вывод отладочной информации обо всех фактах доступа к периферии:

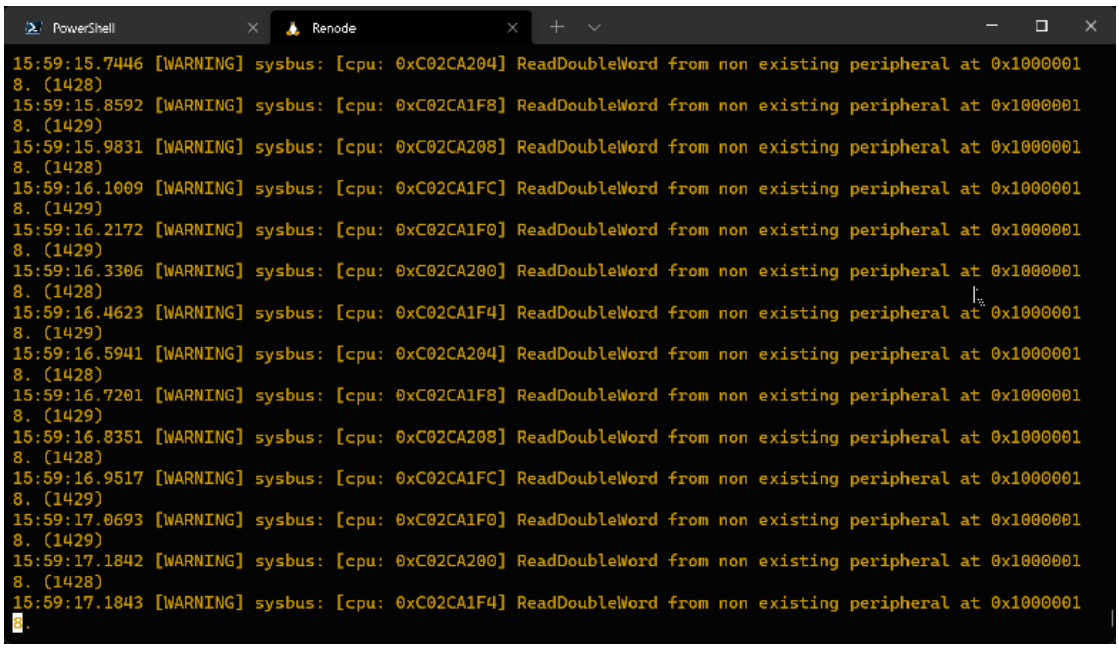
```
# Запустить сервер GDB на localhost:3333
machine StartGdbServer 3333

# Записать 2 инструкции NOP (ARM Thumb) по адресу 0xdeadbeef
sysbus WriteDoubleWord 0xdeadbeef 0x46c046c0

# Писать в консоль обо всех фактах обращения к виртуальным устройствам
sysbus LogAllPeripheralsAccess true
```

Предоставляемых эмулятором Renode возможностей достаточно для быстрого начала работы с исследуемой прошивкой в динамике. Теперь можно запустить эмуляцию:

Запуск эмуляции


В консольном выводе эмулятора мы обнаруживаем, что он, по всей видимости, вошел в бесконечный цикл проверки какого-то условия — значения регистра в памяти — и дальше выполняться не желает:

Бесконечное обращение по адресу 0x1000001




Для начала обращаемся к распакованному Device Tree и пытаемся понять, к какому устройству относится запрашиваемый адрес. Определяем, что по

данному адресу должен находиться Flash Memory Controller — контроллер NOR/NAND-памяти:

```
flash-memory-controller@10000000 {
    compatible = "hisilicon,hisi-fmc";
    reg = <0x10000000 0x1000 0x14000000 0x10000>;
    reg-names = "control\0memory";
    clocks = <0x02 0x2c>;
    #address-cells = <0x01>;
    #size-cells = <0x00>;
...
```

Попробуем обмануть систему, чтобы она смогла продолжить выполнение. Вместо того, чтобы добавлять в файл описания платформы полноценный контроллер Flash-памяти, попробуем эмулировать только запрашиваемый регистр путем добавления тега-заглушки. Но сначала нужно понять, какое значение ожидает ядро в регистре `0x10000018`, чтобы продолжить выполнение. Поиском на GitHub находим драйвер этого контроллера в коде ядра: `drivers/mtd/spi-nor/hisi-sfc.c`, а в нем видим использование искомого регистра в функции `wait_op_finish`:

```
static inline int wait_op_finish(struct hifmc_host *host)
{
    u32 reg;

    return readl_poll_timeout(host->regbase + FMC_INT, reg,
        (reg & FMC_INT_OP_DONE), 0, FMC_WAIT_TIMEOUT);
}
```

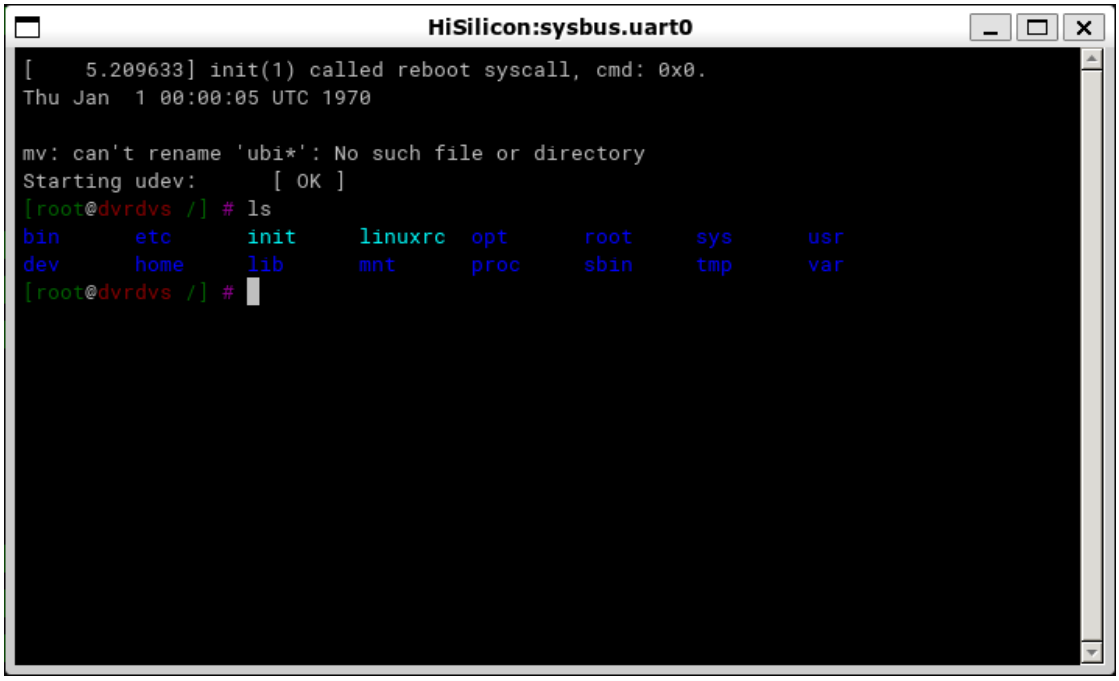
Так как константа `FMC_INT_OP_DONE` равна 1, то для того, чтобы пропустить цикл, нужно, чтобы из регистра `FMC_INT` ядро прочитало 1. Это можно обеспечить, дописав после `Tag <0x120A0000, 0x120A0FFF> "UART2"` в описании платформы следующий тег:

```
Tag <0x10000018, 0x1000001C> "FMCINT" 0x1
```

Теперь по адресу `0x10000018` всегда будет считываться единица. Кроме установки фиксированного значения, Renode позволяет при необходимости реализовывать более сложную логику обработки чтения/записи по адресам с помощью встроенных в описание платформы скриптов на Python. Примеры такого использования можно посмотреть в поставляемых в комплекте с Renode описаниях платформ и скриптах их запуска.

Перезапускаем эмулятор. Теперь в окне терминала, соединенного с виртуальным последовательным портом `uart0`, мы увидим стандартный Linux-шелл, с которым можно взаимодействовать как обычно.

Запуск эмуляции




Таким образом, удалось частично запустить прошивку сетевого видеорегистратора без самого регистратора. Далее с помощью имеющихся на эмулируемой файловой системе утилит можно расшифровать зашифрованные файлы прошивки, извлечь и проанализировать логику работы модулей ядра, обеспечивающих функциональность регистратора, и т. д.

## Обмен данными с эмулируемой системой

Ранее из файла прошивки видеорегистратора нами было извлечено несколько зашифрованных `tar.lzma`-архивов. Беглый анализ присутствующих в запущенной в Renode системе файлов и скриптов показал, что для расшифровки этих архивов используется приложение `/bin/ded`, которое в свою очередь обращается к устройству `/dev/hikded`, предоставляемому модулем ядра `hik_ded.ko`, который в свою очередь обращается к другому модулю ядра `hik_hal.ko`, который уже осуществляет расшифровку ключа шифрования и самого зашифрованного архива. Чтобы не тратить время на детальный анализ всего процесса инструментами статического анализа, можно заставить эту цепочку отработать на зашифрованных файлах в эмуляторе и затем скачать расшифрованные архивы.

Для начала нужно найти способ передать зашифрованные файлы внутрь эмулируемой системы. Из имеющихся устройств, через которые можно было бы организовать взаимодействие, в нашей виртуальной системе на кристалле есть только оперативная память и последовательные порты.

Так как архивы в сумме имеют довольно большой размер, то передавать их через последовательный порт — не самый оптимальный вариант (из-за низкой скорости передачи), поэтому попробуем произвести передачу через оперативную память. Что касается виртуальных последовательных портов, то Renode поддерживает организацию взаимодействия с ними из хост-системы через TCP сервер или создание в ней `pty`-устройства (последний способ работает только на Linux и macOS).

Для того чтобы загрузить в память виртуальной машины зашифрованный файл, используем уже известную команду:

```
sysbus LoadBinary "/home/research/out/sys_app.tar.lzma" 0x48000000
```

Файл `sys_app.tar.lzma` будет загружен в физическую память по указанному адресу. Для того чтобы его оттуда достать, можно использовать устройство `/dev/mem`, которое в Linux позволяет читать и писать непосредственно в физическую память.

```
dd if=/dev/mem of=/sys_app.tar.lzma.encrypted bs=1M seek=1152 count=10
```

Теперь можно выполнить расшифровку штатными средствами прошивки:

```
ded -d /sys_app.tar.lzma.encrypted /sys_app.tar.lzma
tar -atvf /sys_app.tar.lzma
```

После того как мы убедились, что файл успешно расшифровался, встает задача забрать результат расшифровки из виртуальной машины. Сделать это можно также через `/dev/mem`, однако в Renode «из коробки» нет команды, позволяющей сохранить фрагмент содержимого физической памяти в файл, а команды взаимодействия с памятью через GDB-сервер будут оперировать транслированными адресами. Придется реализовать эту возможность с помощью простого плагина:

```
using System;
...

namespace Antmicro.Renode.Plugins.MemoryDumpPlugin {
    public sealed class MemoryDumpCommand : Command {
        public override void PrintHelp(ICommandInteraction writer) {
            base.PrintHelp(writer);
            writer.WriteLine(String.Format("{0} address length \"file\"", Name));
        }

        [Runnable]
        public void Run(ICommandInteraction writer, HexToken address, HexToken lengt
h, StringToken fileName) {
            byte[] memory = monitor.Machine.SystemBus.ReadBytes((ulong)address.Value
, (int)length.Value);
            File.WriteAllBytes(fileName.Value, memory);
        }
```

```
        public MemoryDumpCommand(Monitor monitor) : base(monitor, "dump_memory", "Du
mp memory to file.") { }
    }
}
```

После загрузки этого плагина в консоли Renode появляется новая команда `dump_memory`, с помощью которой можно извлечь из памяти BM необходимые данные и распаковать архив. Дальнейший анализ извлеченных файлов выходит за рамки темы данной статьи, и мы его опускаем.

## Выводы об инструменте

Мы разобрали практический пример использования эмулятора Renode для запуска и расшифровки файлов прошивки реального устройства. В данном случае, благодаря поддержке Renode используемой в SoC HiSilicon периферии, даже не пришлось писать какой-либо код для того, чтобы увидеть полнофункциональный терминал Linux.

В то же время, в случае возникновения такой необходимости, модульная архитектура самого эмулятора и предоставляемые им возможности написания скриптов позволяют с относительно низкими трудозатратами реализовать поддержку отсутствующей функциональности на достаточном для проведения исследования уровне.

Универсальность эмулятора позволяет запускать в нем как полноценные ОС, такие как Linux или QNX, так и небольшие встраиваемые ОСРВ, такие как FreeRTOS, mBed OS, embOS и другие.

Из особенностей данного инструмента стоит отметить его низкоуровневость — эмуляция происходит на системном уровне, поэтому с его использованием сложно провести, например, фаззинг или отладку какого-либо user-space приложения, работающего в эмулируемой операционной системе.

К минусам же можно отнести отсутствие подробной документации — в имеющейся документации описаны лишь самые базовые сценарии работы с инструментом, а в процессе создания чего-то более сложного, как, например, нового периферийного устройства, а также для того, чтобы понять, как работает та или иная встроенная команда, придется неоднократно обращаться к GitHub-репозиторию проекта и изучать исходный код как самого эмулятора, так и существующих реализаций периферийных устройств.

# Отладка на прикладном уровне с помощью GDB и QEMU

Строго говоря, для динамического анализа приложений, работающих на пользовательском уровне, в выбранном нами случае использование каких-либо внешних инструментов не является обязательным: так как исследуемая система — это Linux, то можно ограниченно отлаживать их с помощью самого обычного GDB.

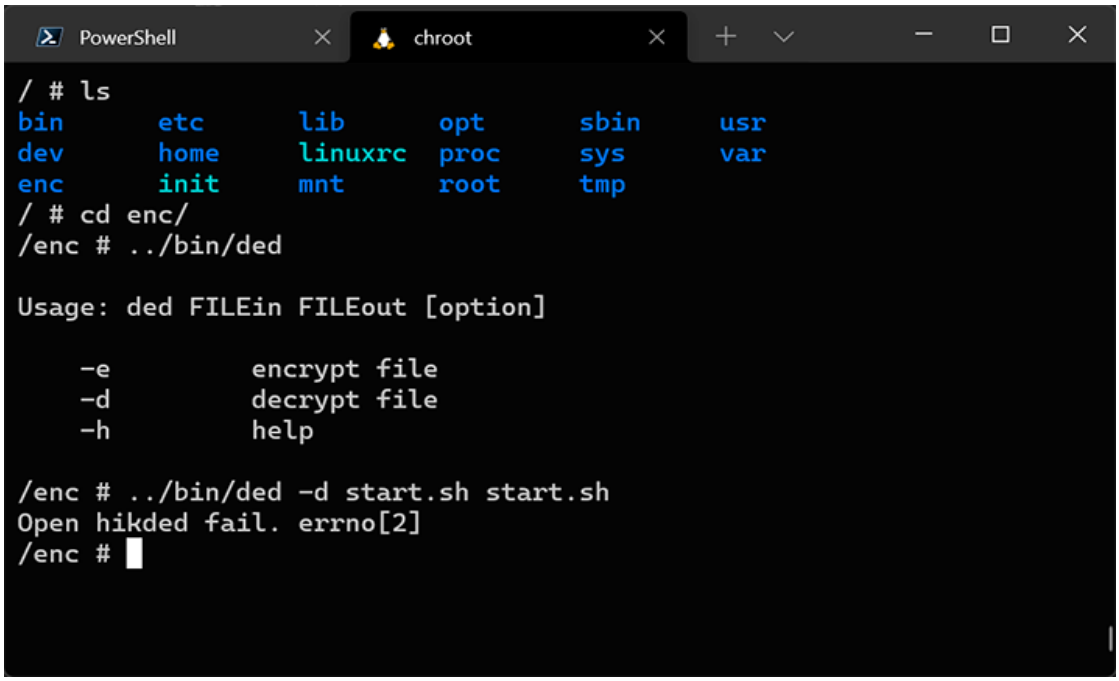
Так как исследуемая прошивка работает на ARM, а исследование проводится на x86-64, то для запуска ELF-файлов потребуется QEMU с поддержкой user-mode ARM эмуляции.

Для проведения эксперимента извлечем из исследуемой прошивки файловую систему, которую мы уже видели в терминале Renode, установим `qemu-user` и `gdb-multiarch`:

```
sudo apt install gcc-arm-linux-gnueabihf libc6-dev-armhf-cross qemu-user-stat
ic gdb-multiarch
```

Теперь можно выполнить команду `chroot` и перейти в окружение, аналогичное тому, которое мы уже видели в Renode. Установленный на предыдущем шаге QEMU обеспечивает прозрачную эмуляцию ARM-инструкций. Особенностью данного подхода является то, что для запуска и отладки в таком окружении будут доступны лишь приложения прикладного уровня, так как ядро Linux в `chroot`-окружении будет унаследовано от основной системы.

chroot в ARM-окружение



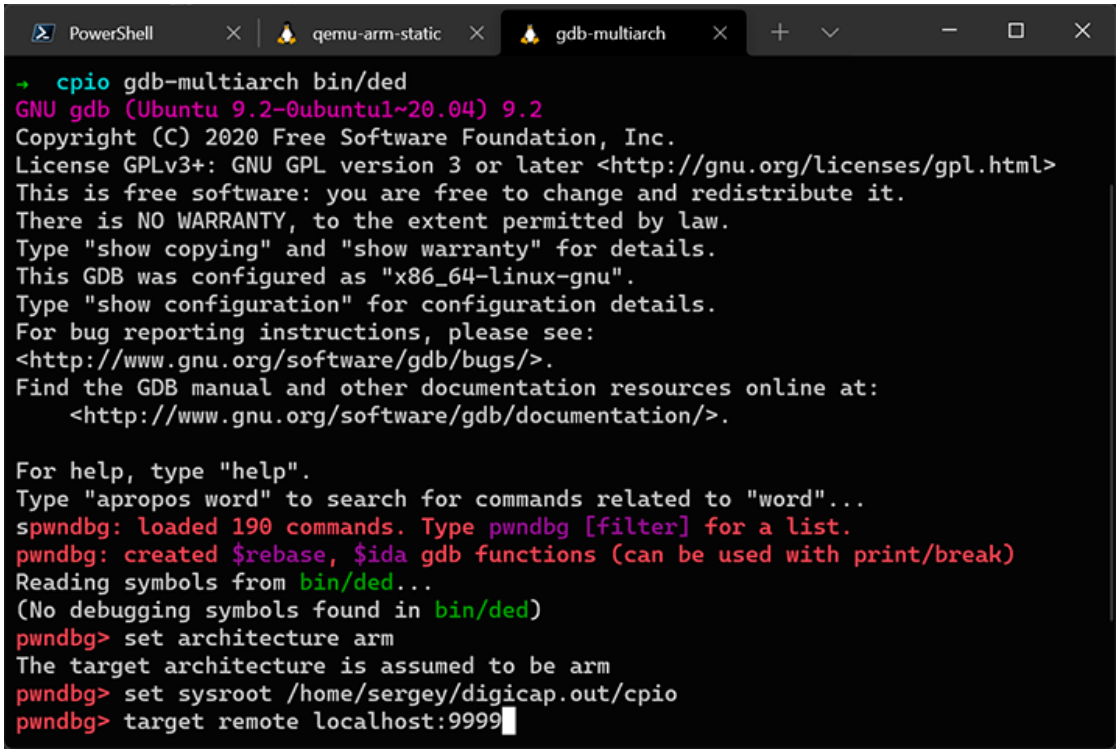

Благодаря встроенному GDB-серверу QEMU позволяет отлаживать Linux-приложения, скомпилированные для архитектуры ARM, с помощью

отладчика GDB на x86-системе. В качестве примера попробуем запустить отладку `/bin/ded`:

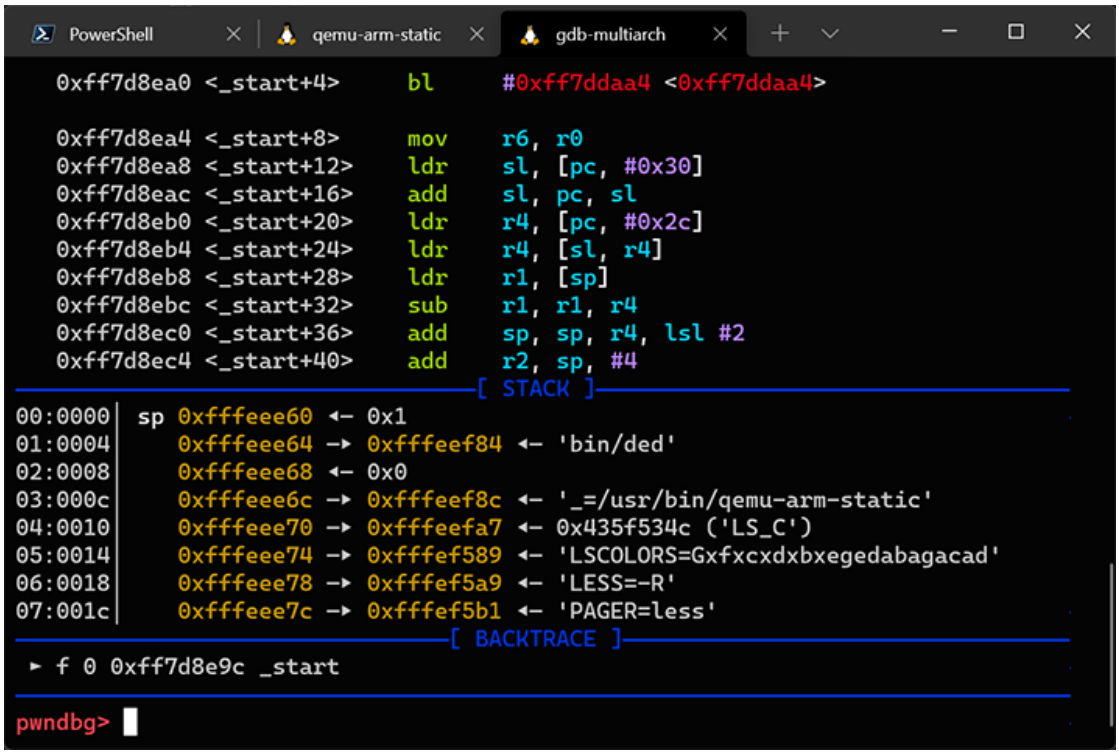
```
qemu-arm-static -g 9999 -L ~/out/cpio/bin/ded
```

После чего в другом терминале запустим GDB:

GDB QEMU 1




GDB QEMU 2




Отладчик запустился и остановился на точке входа.

### Выводы об инструменте

Очевидно, одним из ограничений данного подхода является то, что, в отличие от Renode, при использовании `qemu-user` нет возможности загрузить ядро Linux и его модули, где в исследуемой нами прошивке реализована значительная часть функциональности. В частности, утилита `/bin/ded` из приведенного выше примера выдает ошибку, в то время как в эмуляторе Renode она выполняется без ошибок и корректно расшифровывает зашифрованные файлы. Кроме того, этот подход не будет работать в тех случаях, когда исследуемая прошивка не основана на Linux, так как `qemu-user` работает путем трансляции исполняемого кода ARM в инструкции x86_64 и напрямую транслирует системные вызовы в ядро Linux. Однако следующий способ, который мы рассмотрим, не имеет такого ограничения.

## Отладка на прикладном уровне с помощью Qiling Framework

Qiling — это продвинутый мультиплатформенный фреймворк для эмуляции двоичных файлов, способный эмулировать множество платформ и окружений:

- Эмуляция Windows, MacOS, Linux, QNX, BSD, UEFI, DOS, MBR, Ethereum Virtual Machine;
- Поддержка архитектур x86, x86_64, ARM, ARM64, MIPS, 8086;
- Поддержка различных форматов исполняемых файлов: PE, Mach-O, ELF, COM, MBR.

Сам фреймворк написан на языке Python, что позволяет достаточно легко адаптировать его функциональность под свои нужды. Qiling Framework использует под капотом движок Unicorn для эмуляции. Однако Unicorn — это просто эмулятор машинных инструкций, в то время как Qiling предоставляет высокоуровневые функции, такие как поддержка файловой системы, динамических библиотек, загрузки различных форматов исполняемых файлов и т. д.

В сравнении с QEMU Qiling Framework позволяет эмулировать большее количество платформ, предоставляет возможность гибкой настройки процесса эмуляции, включая, например, изменение выполняющегося кода на лету, и также является кроссплатформенным, то есть позволяет эмулировать, например, исполняемые файлы Windows или QNX на Linux и наоборот. Qiling Framework также содержит в репозитории примеры запуска

фаззера AFL в Unicorn-режиме для исполняемых файлов Linux и QNX, чем мы далее воспользуемся.

Для первого знакомства с Qiling в качестве `hello-world`-примера попробуем запустить с помощью Qiling уже известную утилиту `ded` из исследуемой прошивки. Для этого скопируем файловую систему устройства в `examples/rootfs/hikroot` и напишем простой скрипт `examples/hikded_arm_linux.py`:

```
import sys
sys.path.append("..")

from qiling import Qiling
from qiling.const import QL_VERBOSE

def run_sandbox(path, rootfs, verbose):
    ql = Qiling(path, rootfs, verbose = verbose)
    ql.run()

if __name__ == "__main__":
    run_sandbox(["rootfs/hikroot/bin/ded"], "rootfs/hikroot", QL_VERBOSE.DEFAULT)
```

В процессе работы с уровнем логирования `DEFAULT` Qiling Framework выводит в консоль информацию об эмулируемых системных вызовах, подобно утилите `strace` в Linux.

Теперь можно попробовать запустить фаззер AFL++, который будет использовать Qiling в качестве среды выполнения. Скорость работы такого фаззера в большинстве случаев будет весьма невысокой. Однако благодаря широкой поддержке Qiling Framework эмуляции различных сред и ОС, поддержке UEFI и различных микропроцессорных архитектур, в некоторых случаях это, возможно, будет единственным способом запустить фаззинг с минимальными трудозатратами.

Рассмотренная ранее утилита `ded` не подходит как цель для фаззинга, так как ее код — слишком прост, и она будет завершаться с ошибкой невозможности открытия устройства `/dev/hikded` при любых входных данных. Возьмем для исследования другую утилиту из этой же прошивки — `hrsaverify`. Она предназначена для проверки корректности зашифрованных файлов и принимает в качестве аргумента путь к файлу для проверки. В репозитории Qiling Framework уже имеется несколько примеров запуска фаззера AFL++ в директории `examples/fuzzing`. Для запуска `hrsaverify` адаптируем один из них, а именно `linux_x8664`.

Перепишем скрипт запуска фаззера следующим образом:

```
import unicornafl as UcAfl
UcAfl.monkeypatch()

import os, sys
from typing import Any, Optional

sys.path.append("../../..")
from qiling import Qiling
from qiling.const import QL_VERBOSE
from qiling.extensions import pipe

def main(input_file: str):
    ql = Qiling(["../../rootfs/hikroot/usr/bin/hrsaverify", "/test"], "../../rootfs/
hikroot",
            verbose=QL_VERBOSE.OFF, # keep qiling logging off
            console=False,          # thwart program output
            stdin=None, stdout=None, stderr=None) # don't care about stdin/stdout

    def place_input_callback(uc: UcAfl.Uc, input: bytes, persistent_round: int, data
: Any) -> Optional[bool]:
        """Called with every newly generated input."""
        with open("../../rootfs/hikroot/test", "wb") as f:
            f.write(input)

    def start_afl(_ql: Qiling):
        """Callback from inside."""
        # We start our AFL forkserver or run once if AFL is not available.
        # This will only return after the fuzzing stopped.
        try:
            if not _ql.uc.afl_fuzz(input_file=input_file,
                                   place_input_callback=place_input_callback, exits=
[ql.os.exit_point]):
                _ql.log.warning("Ran once without AFL attached")
                os._exit(0)

        except UcAfl.UcAflError as ex:
            if ex.errno != UcAfl.UC_AFL_RET_CALLED_TWICE:
                raise

    # Image base address
    ba = 0x10000
    # Set a hook on main() to let unicorn fork and start instrumentation
    ql.hook_address(callback=start_afl, address=ba + 0x8d8)
    # Okay, ready to roll
    ql.run()

if __name__ == "__main__":
    if len(sys.argv) == 1:
        raise ValueError("No input file provided.")

    main(sys.argv[1])
```

Первое, на что следует обратить внимание — это базовый адрес исполняемого файла (в нашем случае `0x10000`), адрес функции `main`. Иногда бывает необходимо дополнительно установить перехватчики на другие адреса, переход на которые фаззеру следует засчитать как падение.

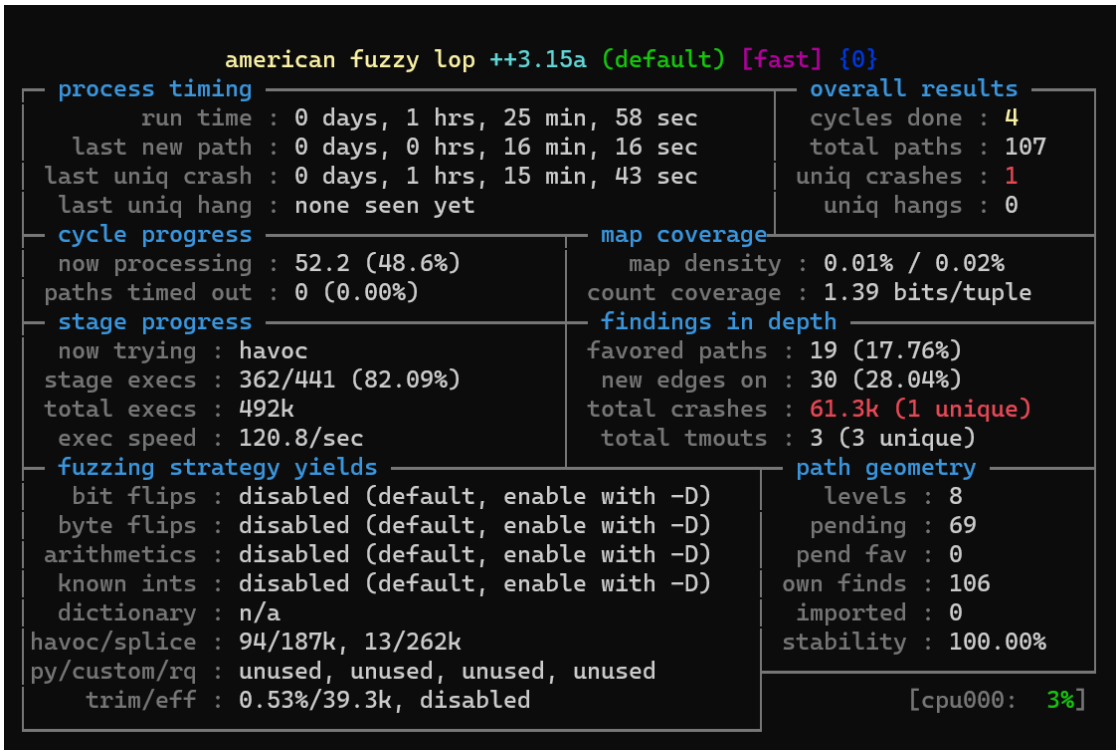
Например, в примере запуска AFL в среде QNX (в директории `qnx_arm`) устанавливается такой дополнительный обработчик на адрес функции `SignalKill` в `libc`. В случае с `hrsaverify` можно обойтись без дополнительных обработчиков. Также следует обратить внимание, что все файлы, которые должны быть доступны для выполняемого приложения, следует помещать в `sysroot` и передавать пути к ним относительно него (в данном случае это `../../rootfs/hikroot/`).

Непосредственно запуск AFL++ выполняется следующей командой:

```
AFL_AUTORESUME=1 AFL_PATH="$(realpath ./AFLplusplus)" PATH="$AFL_PATH:$PATH" afl-fuz
z -i afl_inputs -o afl_outputs -U -- python ./fuzz_arm_linux.py @@
```

В результате будет запущен фаззер AFL, и через некоторое время мы увидим результат его работы:

AFL++




## Выводы об инструменте

Qiling — перспективный инструмент, основными плюсами которого является высокая гибкость, легкая расширяемость, поддержка большого количества архитектур и окружений. Фреймворк может заменить `qemu-user` в случаях, когда использование последнего не представляется возможным (неподдерживаемая целевая ОС или отсутствие необходимых дополнительных возможностей, таких как установка произвольных обработчиков на любые адреса памяти, особая обработка прерываний и т. д.).

Платой за высокую гибкость и низкий порог вхождения за счет использования языка высокого уровня Python является низкая скорость эмуляции.

## Бонус: отладка на реальном устройстве с помощью GDB

На платах многих устройств разработчики умышленно оставляют различные служебные интерфейсы, такие как UART-консоль, JTAG и т. д. Часто данные интерфейсы деактивированы на программном или аппаратном уровне, защищены паролем, либо доступ к ним тем или иным образом затруднен. Однако нередки случаи, когда эти интерфейсы доступны изначально, или исследователю удаются получить к ним доступ в процессе работы. В таких случаях открывается возможность взаимодействия с исследуемой системой через командную строку или JTAG-отладчик.

В случае JTAG можно считать или записать содержимое в ПЗУ или оперативную память, установить точки останова, управлять периферийными устройствами путем записи в соответствующие им адреса регистров ввода-вывода, анализировать состояние регистров процессора и т. д.

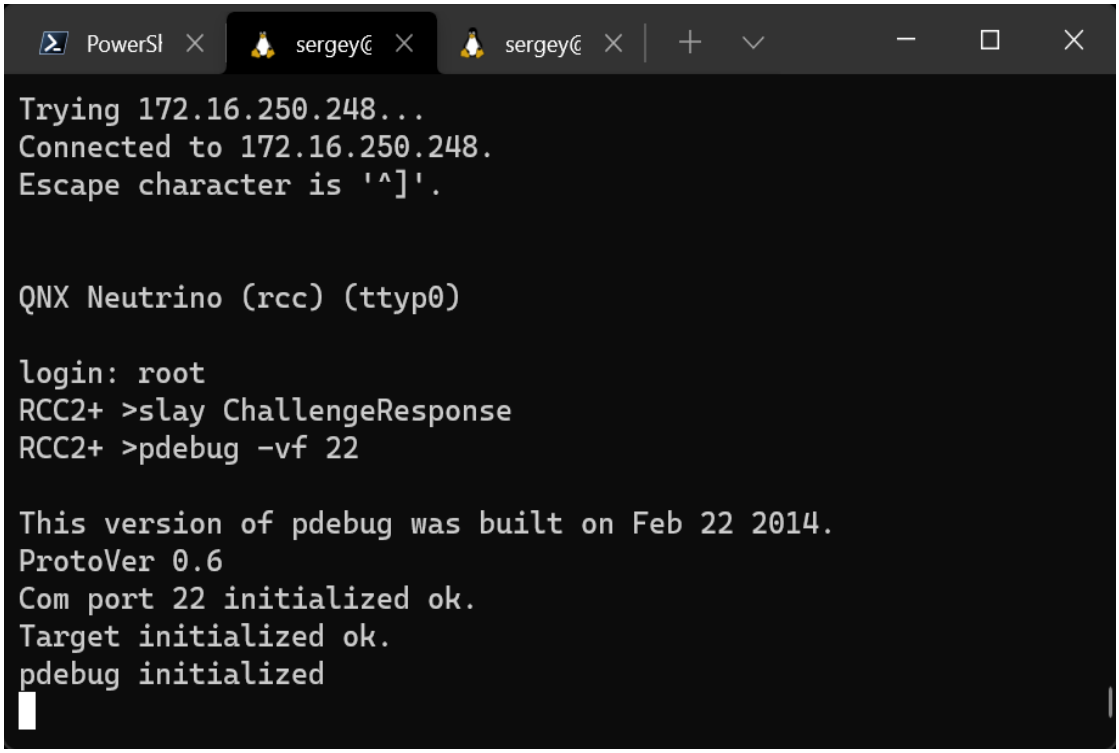
В случае с UART часто бывает возможно взаимодействовать с начальным загрузчиком (например, U-Boot), а также командной оболочкой загруженной операционной системы. В сочетании с возможностью модификации содержимого ПЗУ либо доступностью иных способов записи/редактирования файлов в файловой системе устройства (например, загрузки файлов по сети через FTP), это открывает возможность интерактивной отладки приложений прямо на устройстве. Обычно, чтобы воспользоваться этим способом, сначала требуется скомпилировать сервер GDB для целевой архитектуры и ОС.

Для демонстрации этого подхода выполним подключение отладчика к головному устройству (head unit) мультимедиа системы для автомобилей одного из наиболее популярных вендоров, которое работает под управлением ОС реального времени QNX.

В состав SDK QNX входит модифицированный отладчик GDB с собственным протоколом удаленной отладки, несовместимым со стандартным GDB-сервером. Поэтому вместо сервера GDB на устройстве необходимо запустить демон `pdebug`, предварительно скопировав его из SDK, если его нет на устройстве. Демон может взаимодействовать с удаленным отладчиком GDB через TCP или последовательный порт, соответственно, для отладки необходимо, чтобы как минимум один из этих способов коммуникации с устройством был доступен.

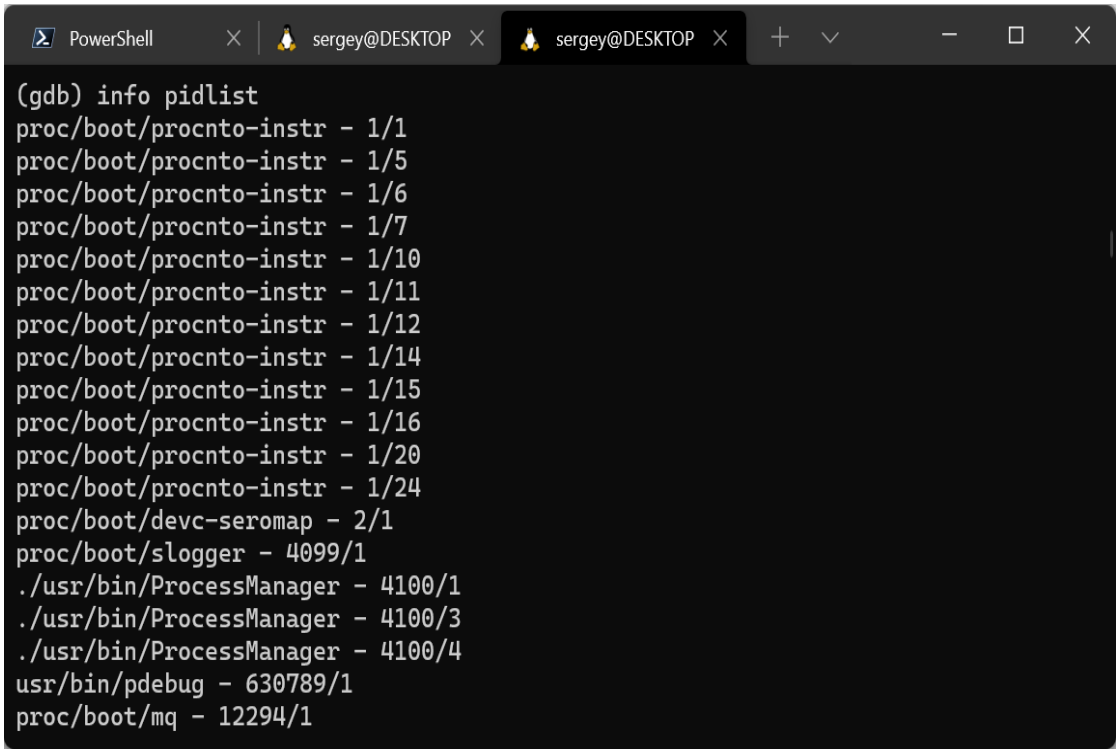
Путем модификации содержимого памяти в файловую систему устройства нами был добавлен демон `pdebug`. Запустим его:

Запуск демона pdebug



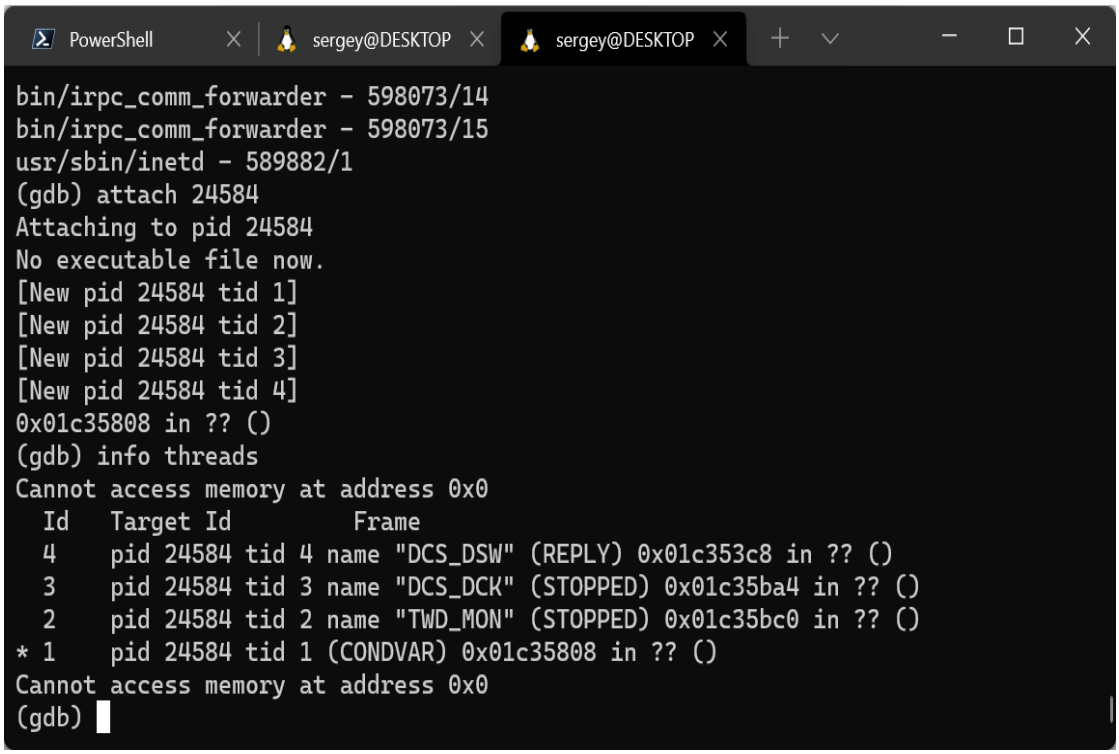

В качестве порта для подключения используется TCP-порт 22, который в исследуемом устройстве доступен к подключению извне, в то время как все остальные порты защищены брандмауэром. Теперь к демону можно подключиться с помощью отладчика GDB, входящего в состав QNX Software Development Platform (SDK), и выбрать процесс для отладки из списка уже запущенных процессов, либо запустить новый.

GDB подключен к головному устройству по TCP




С кратким руководством по отладке с использованием GDB в QNX можно ознакомиться в официальной документации.

GDB подключен к процессу с идентификатором 24584




Кроме использования непосредственно самого исследуемого устройства для отладки можно также использовать другие подходящие устройства.

Например, рассматриваемую в данном примере ОС QNX 6.6 можно также запустить на легко доступной на рынке плате BeagleBone Black. Хотя по аппаратному составу она будет отличаться от исследуемого устройства, некоторые приложения из прошивки вполне вероятно можно будет исследовать таким образом, скопировав на частично совместимую плату и запустив `pdebug` на ней.

Это может быть полезно, так как модификация прошивки устройства с целью загрузки в нее недостающего демона `pdebug`, продемонстрированная в этом разделе, потребовала много времени и усилий. В то же время скопировать отдельные файлы из прошивки и запустить их на совместимом процессоре может оказаться более выгодным решением с точки зрения затрачиваемых ресурсов.

### Выводы об инструменте

Как правило на начальных этапах исследования доступ к различного рода отладочным интерфейсам промышленных устройств затруднен или невозможен, поэтому, вероятно, потребуется приложить усилия, чтобы этот доступ обеспечить — активировать отключенные интерфейсы, впаять недостающие электронные компоненты, модифицировать прошивку и т. д. Однако после того, как эта промежуточная задача решена, отладка непосредственно на самом устройстве может сильно облегчить дальнейший процесс исследования.

## Заключение

Разумеется, в статье не показано всё разнообразие проблем, с которыми автору приходится сталкиваться в процессе исследования прошивок устройств, однако он постарался выбрать те из них, которые наиболее часто приходится решать в самом начале исследования, и которые, по мнению автора, могут стать камнем преткновения для начинающего исследователя.

Автор надеется, что приведенный обзор инструментов упростит начинающим исследователям их работу и ускорит дальнейшее самостоятельное погружение в тему динамического анализа кода прошивок устройств.

Автор также надеется, что более опытных исследователей обзор подтолкнет к мысли об участии в дальнейшем развитии описанных утилит и фреймворков.

*Если вам нужно больше информации или вы хотите поделиться своими мыслями, напишите автору или на адрес ics-cert@kaspersky.com.*

Центр реагирования на инциденты информационной безопасности промышленных инфраструктур «Лаборатории Касперского» (Kaspersky ICS CERT) — глобальный проект «Лаборатории Касперского», нацеленный на координацию действий производителей систем автоматизации, владельцев и операторов промышленных объектов, исследователей информационной безопасности при решении задач защиты промышленных предприятий и объектов критически важных инфраструктур.

Kaspersky ICS CERT ics-cert@kaspersky.com