LoRaWAN’s Protocol Stacks: The Forgotten Targets at Risk

Technical Brief

LoRaWAN technology is used across different industries to monitor critical applications. Usually, these small devices connect sensors with a network. For example, many industrial facilities rely on these sensors to keep an eye on smoke, fire, flood, or weather conditions. The devices are also used in modern connected cities to make environments smart in a way that reduces maintenance costs and improves quality of life.

An attack affecting these devices could have damaging effects on property and even users, depending on how the technology is integrated into the environment. Attacks could lead to out-of-control factory issues, sensitive data leaks, or many other dangerous security scenarios.

In our previous publications[1,2,3], we talked about the known LoRaWAN entry vectors that attackers usually target. The LoRaWAN stack is not a vector that is usually included in conversations about LoRaWAN security, but it is actually the root of LoRaWAN implementation — and so of its security. An attack on the stack could have severe consequences.

In this report, we show the techniques that an attacker can use to find exploitable flaws in the LoRaWAN stack. We bring these details forward to highlight that the same techniques can be used by stack developers or a security consultant to secure the stack and make LoRaWAN communication resistant to critical bugs.

Introduction

Although we have cited significant security issues and practices in our previous publications about LoRaWAN,[4] there are still areas concerning the implementation of the LoRaWAN stacks used by connected devices that are in the dark. Most of these security issues are about the confidentiality and integrity of data. The exploitation of a protocol stack vulnerability would allow an attacker to execute malicious code on target devices, which in turn could have compounded security effects depending on the target and its capabilities.

In most existing publications, protocol stack topics are exceedingly rare. Because of this, we thought that making this report on the LoRaWAN stack could complete our security series. Here, we discuss LoRaWAN stack implementation and how to hunt for bugs in the different stacks using different techniques, such as fuzzing with AFL++.[5] In the section about fuzzing, we introduce our fuzzing platform, which includes several harness tools that help us during the fuzzing process. In the next section, we speak about emulation using Qiling (based on Unicorn Engine)[6] with respect to fuzzing and debugging exotic architectures. In addition to Qiling, we also discuss an alternative method using Ghidra's PCode emulation, which is done when targeted architectures are not supported by Unicorn or Qiling.

We hope the discussion on these techniques will help security teams include protocol stack security testing in the Deming wheel and avoid risks of compromise.

The LoRaWAN Stack

There are at least two types of stacks we can find with LoRaWAN: an end-node stack and a gateway stack.

It should be noted that there are also other stacks for the network and application servers, but we will focus on vulnerabilities that we can trigger from the radio interface. Indeed, an attacker has a better chance of accessing the radio interface rather than the network because it is more exposed. It is also worth noting that attackers from the radio side will act as a kind of malicious sensor or gateway.

End-node stack

The end-node stack is implemented in end-node devices as well as used to send uplink (UL) packets to the gateway. But on some occasions, the network can also send information to the end-node devices, and so end-node stacks also need to manage downlink (DL) packets forwarded by the gateway.

The following figure shows how end-node devices are placed in a typical LoRaWAN infrastructure:

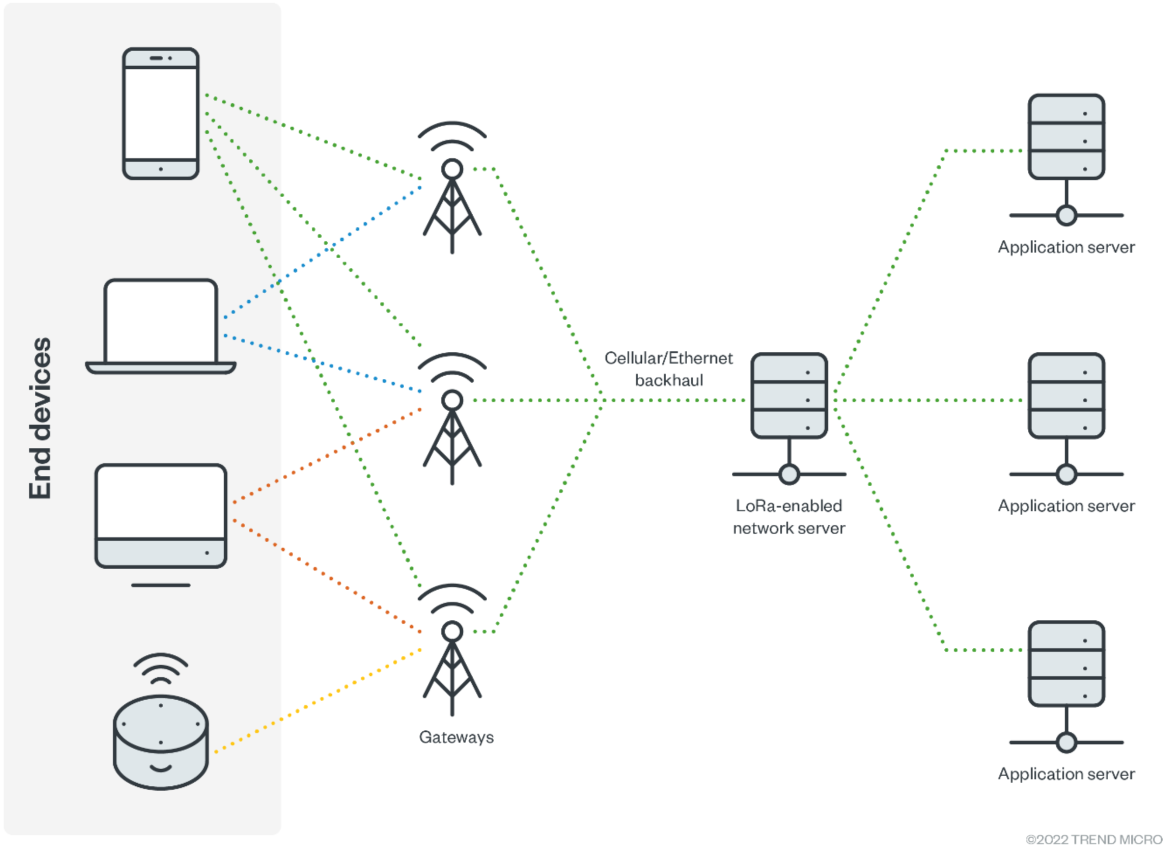

Figure 1. LoRaWAN network architecture

DL packets include the join procedure in the end-node device side, as well as data that comes from the network. For example, the open-source implementation from Semtech called "LoRaMac-node"[7] implements the different types of packets that come from the network and are forwarded by the gateway. This is shown in the following lines in C from */src/mac/LoRaMac.c* sources:

```
static void ProcessRadioRxDone( void )
{
[…]
switch( macHdr.Bits.MType )
    {
        case FRAME_TYPE_JOIN_ACCEPT:
[…]
        case FRAME_TYPE_DATA_CONFIRMED_DOWN:
            MacCtx.McpsIndication.McpsIndication = MCPS_CONFIRMED;
            // Intentional fall through
        case FRAME_TYPE_DATA_UNCONFIRMED_DOWN:
[…]
}
```

In addition to the direction of chips in radio, therefore, only packets of those types are expected by the end-node device.

Gateway stack

The gateway stack includes all functions to connect to the network and forwards packets that come from it. It can also forward packets coming from the end-node to the network using the radio interface. An example of a gateway implementation can be found with the open-source LoRa Basics Station[8] as follows:

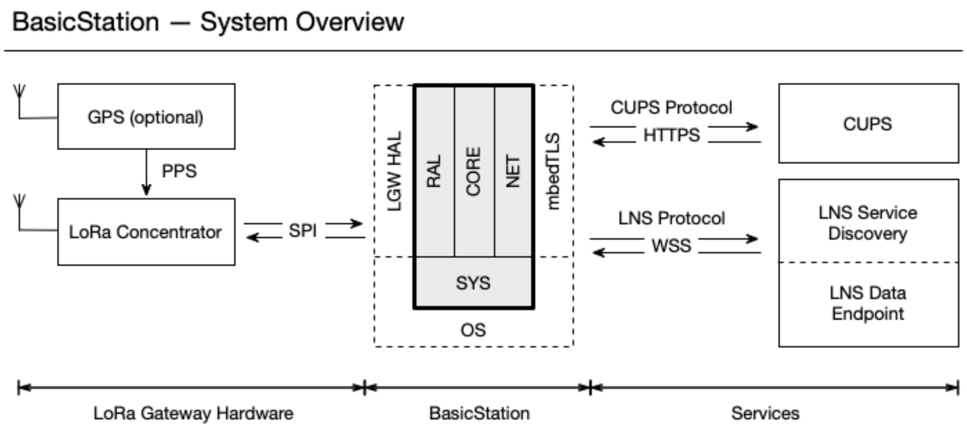

Figure 2. BasicStation architecture for gateways

By default, the gateway is not an important target for an attacker who wants to use the radio interface. Generally, gateways only forward the packet to the core without interpreting them. On rare occasions, we can see custom gateways capable of interpreting packets; this means that though they are possible targets, they are not typically so. Nevertheless, some implementations can be configured in standalone mode (thereby avoiding the connection to a LoRaWAN network), such as TheThingsNetwork (TTN),[9] which will soon be completely upgraded to The Things Stack V3.[10]

Indeed, Dragino is one of the vendors allowing the gateways to be put in standalone mode using Authentication By Personalization (ABP) mode:

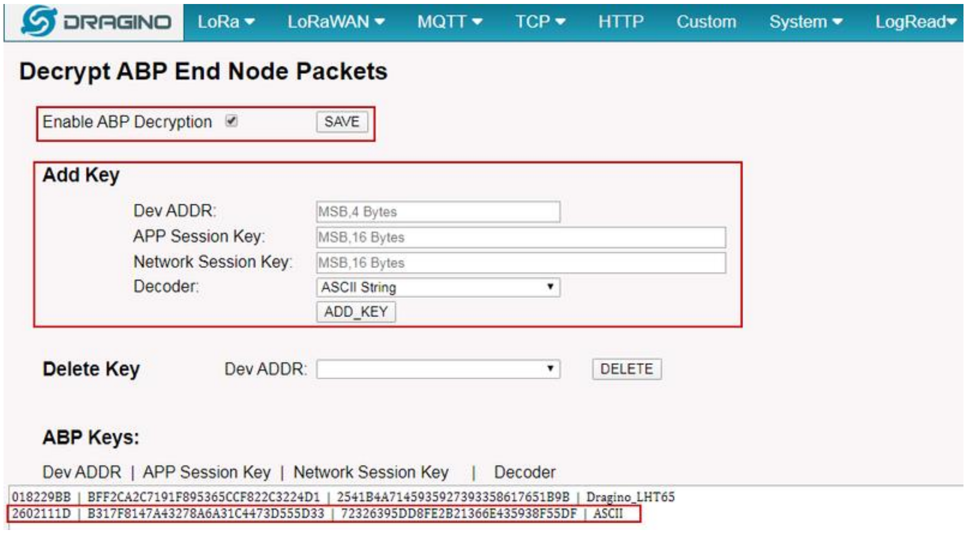

Figure 3. Enabling ABP end-node packets decryption on Dragino’s gateway

Enabling end-node packet decryption in ABP mode implies that the gateway will parse the packets at some point. This can be seen with static analysis of the *lora_pkt_fwd* binary from the Dragino LG308 gateway:

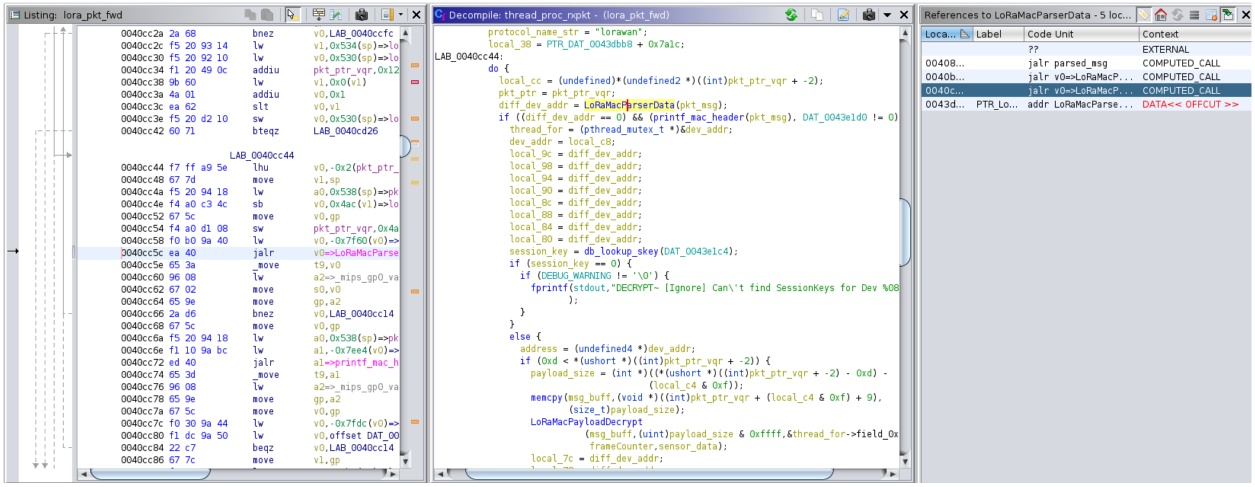

Figure 3. Analyzing *lora_pkt_fwd* of an LG308 in MIPS MSB with Ghidra

Dragino's binary-like *lora_pkt_fwd* implementation is based on Semtech's code but was customized to introduce packet parsing and decryption. The code can be found in one of Dragino's Github repositories by digging into their packages for LG308 devices.[11]

Finding Bugs in the Packet's Parser

In the second article in our LoRaWAN series,[12] we showed a method of using Scapy not just to parse but also to generate LoRaWAN packets. This way, it was also possible to see that LoRaWAN packets have many fields that could be badly implemented.

There are several ways to find bugs in the protocol stacks of LoRaWAN and other protocols:

- Using statical analysis
- Running fuzzing campaigns
- Finding them accident (our favorite)
- Using hybrid approaches

The benefit of statistical analysis is that we are more precise about the presence of bugs and their nature. However, sometimes we must then spend a considerable amount of time trying to understand closed source code and the interesting path of code we are analyzing is never hit. In other words, it's like looking for a needle in a haystack. Fortunately, a technique called "fuzzing" exists to perfect the task.

Fuzzing consists of generating and mutating inputs that we will feed into our program to find bugs. This technique is derivate of accidental bug finding since we introduce bad input data into our program that our parser is not supposed to handle.

There are many ways to generate LoRaWAN packets:

- **"Dumb":** Bit-flipping using a valid packet
- **"Smart":** Using a generator like our *loraphy2wan* Scapy layer[13]
- **Solver-based generation:** Using Satisfiability Modulo Theories (SMT) solver such as z3, or even frameworks like Triton[14] to generate input payloads

The "dumb" way is not the fastest way to generate valid payloads, but it takes less time to write a dumb fuzzer than a "smart" generator, which needs to understand the structure of a packet. When generators can produce a finite result of test cases depending on how elements are handled, dumb fuzzing can be powerful in finding bugs that a standard generator can miss.

Instrumenting the packet-parsing process

Before fuzzing anything, we need to compile the code into something easily handled by a fuzzer. Indeed, as we have the source code, we will not compile the code and fuzz in radio as there will be a lot of introduced latencies. Instead, we will change the code in such a way that it will directly process given "packets."

When we look at cross references to the *LoRaMacParserData()* function deeper, we see that it is called in *LoRaMacCryptoUnsecureMessage()* as well as *ProcessRadioRxDone()*, but *LoRaMacCryptoUnsecureMessage()* itself is called in *ProcessRadioRxDone(),* so the conclusion is that everything begins at *ProcessRadioRxDone()*.

ProcessRadioRxDone() is called by an Interrupt ReQuest (IRQ) that we can track using a cross-reference engine as follows:

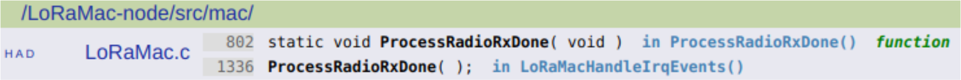

Figure 4. LoRaMAC-node sources browsing on OpenGrok

The following RxDone callback functions will handle IRQ events from the LoRa radio driver:

```
static void LoRaMacHandleIrqEvents( void )
{
    LoRaMacRadioEvents_t events;

    CRITICAL_SECTION_BEGIN( );
    events = LoRaMacRadioEvents;
    LoRaMacRadioEvents.Value = 0;
    CRITICAL_SECTION_END( );

    if( events.Value != 0 )
    {
        if( events.Events.TxDone == 1 )
        {
            ProcessRadioTxDone( );
        }
        if( events.Events.RxDone == 1 )
        {
            ProcessRadioRxDone( );
        }
        if( events.Events.TxTimeout == 1 )
        {
            ProcessRadioTxTimeout( );
        }
        if( events.Events.RxError == 1 )
        {
            ProcessRadioRxError( );
        }
        if( events.Events.RxTimeout == 1 )
        {
            ProcessRadioRxTimeout( );
        }
    }
}
```

Figure 5. LoRaWAN IRQ event handler

Therefore, if we want to perform fuzzing tests independently from these IRQ, we can start directly from the *ProcessRadioRxDone()* function.

This function, however, does not use any argument to pass, so we must find out how to pass the packet message for processing. Luckily, the function is not extraordinarily complex, and we can quickly figure out what variable will be necessary to use for our fuzzing:

```
static void ProcessRadioRxDone( void )
{
    LoRaMacHeader_t macHdr;
    ApplyCFListParams_t applyCFList;
    GetPhyParams_t getPhy;
    PhyParam_t phyParam;
    LoRaMacCryptoStatus_t macCryptoStatus = LORAMAC_CRYPTO_ERROR;

    LoRaMacMessageData_t macMsgData;
    LoRaMacMessageJoinAccept_t macMsgJoinAccept;
    uint8_t *payload = RxDoneParams.Payload;
    uint16_t size = RxDoneParams.Size;
    int16_t rssi = RxDoneParams.Rssi;
    int8_t snr = RxDoneParams.Snr;
[…]
```

The *RxDoneParams* is also a global structure (part of *LoRaMac-node/src/mac/LoRaMac.c*), and we can change it on the fly to fill with our custom payload:

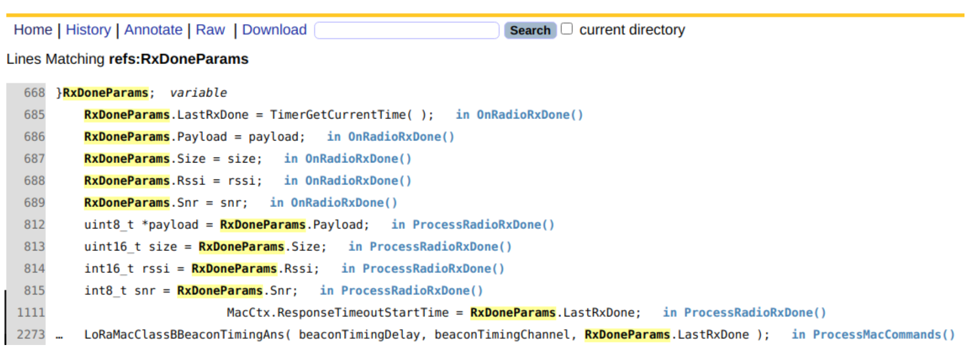

Figure 6. RxDoneParams' structure

Using this structure, we can initialize our payload on the fly in a *main()* function that will take the payload as an argument and call the *ProcessRadioRxDone*.

When trying to compile all the sources, there are still some timers and/or schedulers specific to the used microcontroller unit (MCU). The original *ProcessRadioRxDone()* function will need to be copied and the native functions commented out to compile in the targeted architecture when we want to fuzz.

In copying this *ProcessRadioRxDone()* function we need to include its dependencies and comment architecture-specific function calls as follows:

```
static void ProcessRadioRxDone( void )
{
    LoRaMacHeader_t macHdr;
[...]

    //Radio.Sleep( );
    //TimerStop( &MacCtx.RxWindowTimer2 );

    // This function must be called even if we are not in class b mode
yet.
    /*if( LoRaMacClassBRxBeacon( payload, size ) == true )
    {
        MacCtx.MlmeIndication.BeaconInfo.Rssi = rssi;
        MacCtx.MlmeIndication.BeaconInfo.Snr = snr;
        return;
    }*/
    // Check if we expect a ping or a multicast slot.
    /*if( MacCtx.NvmCtx->DeviceClass == CLASS_B )
    {
        if( LoRaMacClassBIsPingExpected( ) == true )
        {
            LoRaMacClassBSetPingSlotState( 0 );
            LoRaMacClassBPingSlotTimerEvent( NULL );
            MacCtx.McpsIndication.RxSlot =
RX_SLOT_WIN_CLASS_B_PING_SLOT;
        }
        else if( LoRaMacClassBIsMulticastExpected( ) == true )
        {
            LoRaMacClassBSetMulticastSlotState( 0 );
            LoRaMacClassBMulticastSlotTimerEvent( NULL );
            MacCtx.McpsIndication.RxSlot =
RX_SLOT_WIN_CLASS_B_MULTICAST_SLOT;
        }
    }*/

    macHdr.Value = payload[pktHeaderLen++];
     switch( macHdr.Bits.MType )
    {
```

It should be noted that we have also removed calls to modes we are not using by default to simplify the task.

Moreover, some other initialized context is needed to avoid any unwanted crashes happening when parsing, deciphering the messages, or processing missing queues in the code. This results are in the following *main()* function:

```
void main(int argc, char *argv[])
{
    LoRaMacCryptoNvmEvent * cryptoNvmCtxChanged;
```

```
    FILE *fp;
     char buff[500]; // larger enough to mess with that

    if (argc == 2)
    {
        fp = fopen(argv[1], "r");
        fgets(buff, 256, (FILE*)fp); // a little big than the MAX size of a LoRaWAN packet, but let’s find some bugs on the code...

        LoRaMacCryptoInit(cryptoNvmCtxChanged);
       MacCtx.NvmCtx = malloc(sizeof(LoRaMacNvmCtx_t));
       MacCtx.MacCallbacks = malloc(sizeof(LoRaMacCallback_t));
       LoRaMacInitialization2( ); // a derivate function without architecture specific calls
       RxDoneParams.Payload = buff;
       RxDoneParams.Size = strlen(buff);
       ProcessRadioRxDone();
    }
}
```

After compiling a code that will process a packet provided in the argument line, we can start thinking about how we can process it when fuzzing this stack.

Fuzzing the stack

Proposed design

For our purpose, we will combine the generation that will allow us to cover as many code paths as possible with legitimate and dumb fuzzing using the AFL++ framework (evolution of AFL) that supplies some instrumentation for pseudorandomly mutating the bits, bytes, words.

First, we generate and capture legitimate packets coming from the LoRaWAN network; these are mostly downlink packets since we are studying the end-node stack. Captured and generated packets will then be saved in independent byte files that will be reduced using a minimizer that will filter the input that is useless to mutate (based on the code path coverage). Essential inputs will be fed to an AFL++ fuzzer that instruments a different binary base on the strategy and produces crashes. Produced crashes are then classified by the type of vulnerability and its backtrace, and then they are minimized to the smallest useful payload that can be debugged.

We explain the main points of this architecture in the following sections. Figure 7 shows the whole architecture that we have designed for our fuzzing tests:

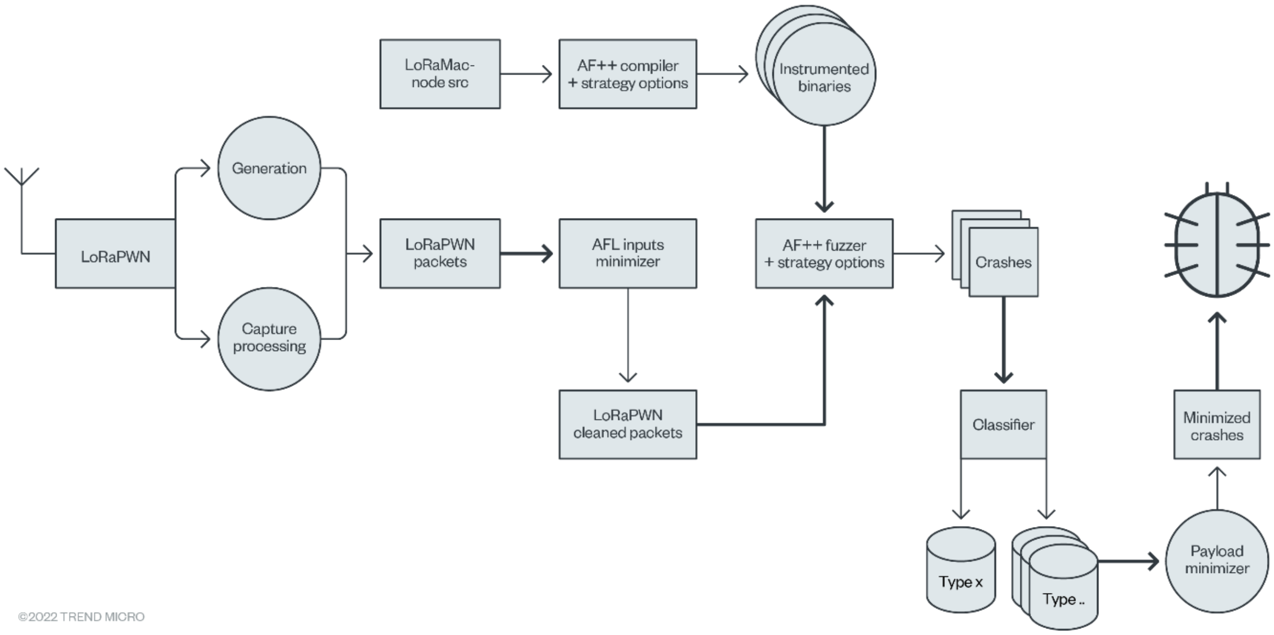

Figure 7. Fuzzing architecture design for radio protocol layer as applied to LoRaWAN

Generation and captures

To cover as much code path as possible, we needed to collect every type of message that could be interpreted by the parser. The first approach consists of capturing messages using our LoRaPWN framework, which works as follows:

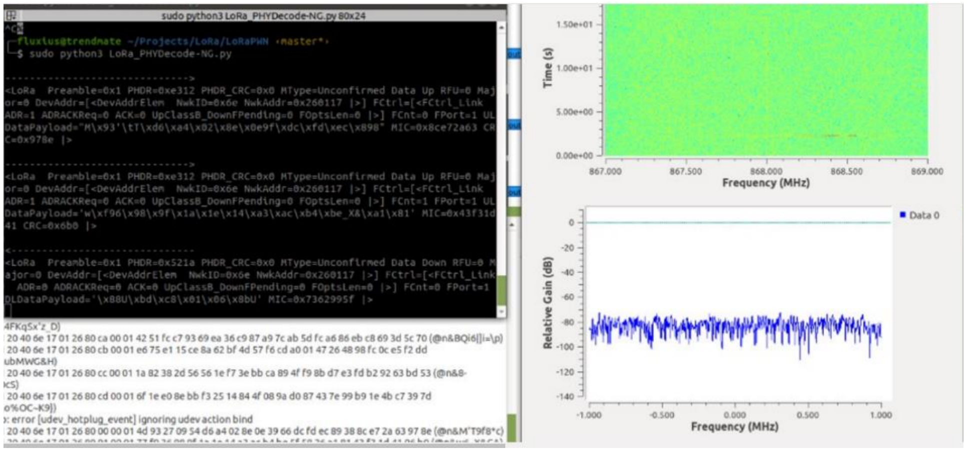

Figure 8. Capturing and processing packets with the LoRaPWN framework

We can then arrange or generate the captured packet using the interactive mode of the framework:

```
~>>> ABPpkt
<LoRa Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up
RFU=0 Major=0 DevAddr=[<DevAddrElem NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=9 FPort=1
ULDataPayload='\x9ec)Egc\xdb\x9a\x8cT\xde\xd3wF\xa9\xce\xc8'
MIC=0xecc33bc4 CRC=0x922 |>

~>>> ABPpkt.MType=0
~>>> ABPpkt
<LoRa Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Join-request RFU=0
Major=0 Join_Request_Field=None MIC=0xecc33bc4 CRC=0x922 |>
…
```

Binary instrumentation and strategies

Fuzzing is a very long process. We must use as much CPU as possible to parallelize the work and gain some time, or try different strategies to trigger as many bugs and/or crashes as possible. For that, AFL++ allows us to use main and secondary fuzzers with “-M” and “-S” options:

```
afl-fuzz -M 01 -i finput -o fout -- ./Fuzzy2 @@
```

For secondary fuzzers, it is better to use variations, unless we want to fuzz the exact same thing. AFL++ allows interesting variation compilations, as listed here:

- With sanitizers activated (*export AFL_USE_ASAN=1 ; export AFL_USE_UBSAN=1 ; export AFL_USE_CFISAN=1;*)
- CMPLOG/redqueen
- laf-intel/COMPCOV

Other secondary sessions could also be run, such as

- A third- to a half-session with the MOpt mutator enabled, -L 0
- Using different a power schedule, like explore (default), fast, coe, lin, quad, exploit, mmopt, rare, and seek (for example, -p seek)

Optimizing fuzzing with persistent mode

The persistent mode is used to increase the fuzzing process speed from by x2 to by x20. Using this mode, the fuzzer feeds test cases in separate long-lived processes, avoiding costs when *fork()*ing the program.

We have performed the following changes to use the persistent mode in the **main.c** binary:

```
18a19
> #include <limits.h>
2130a2132
> __AFL_FUZZ_INIT();
2133a2136
>
2137a2141,2146
> #ifdef __AFL_HAVE_MANUAL_CONTROL
>   __AFL_INIT();
> #endif
>
> while (__AFL_LOOP(UINT_MAX)) {
>
2141,2142c2150,2151
< fp = fopen(argv[1], "r");
< fgets(buff, 256, (FILE*)fp);
---
> //fp = fopen(argv[1], "r");
> //fgets(buff, 256, (FILE*)fp);
2155,2157c2164,2170
< RxDoneParams.Payload = buff;
< RxDoneParams.Size = strlen(buff);
< ProcessRadioRxDone();
---
> RxDoneParams.Payload = __AFL_FUZZ_TESTCASE_BUF;
> RxDoneParams.Size = __AFL_FUZZ_TESTCASE_LEN;
> if (RxDoneParams.Size < 256)
> {
> ProcessRadioRxDone();
> }
> }
```

Note that this mode is not as stable as the standard mode. That is why we keep to different versions of instrumented **main.c** source code.

Classifications

The classification part can be considerably helpful when dealing with the many "uniq crash files" found in a repository:

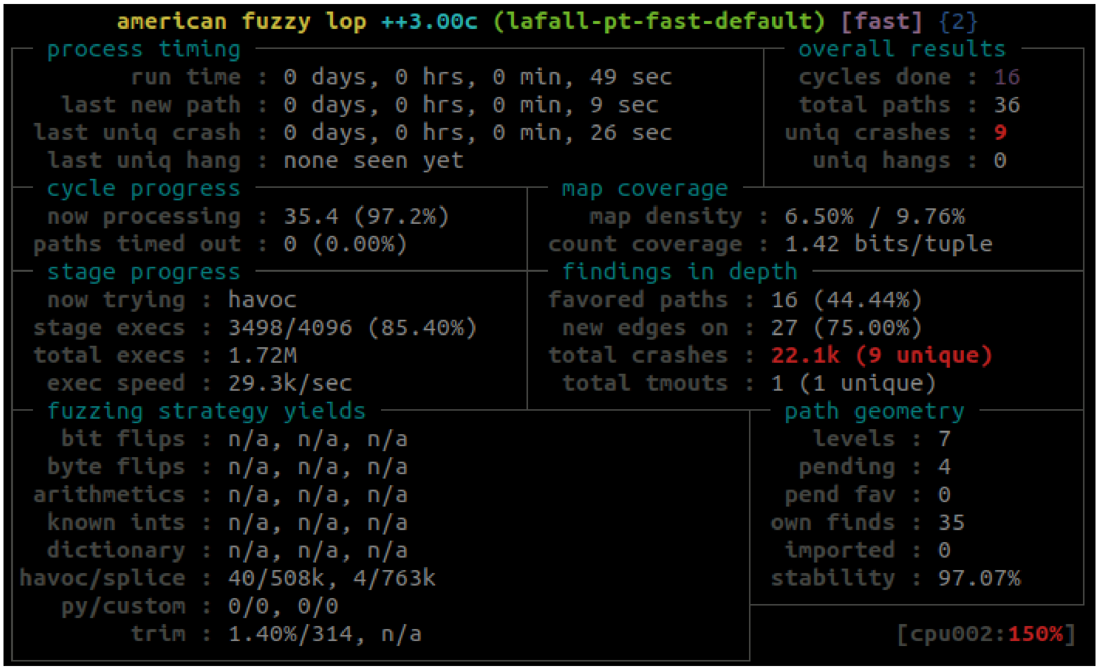

Figure 9. Example of a fuzzing session on LoRaMAC-node with AFL++

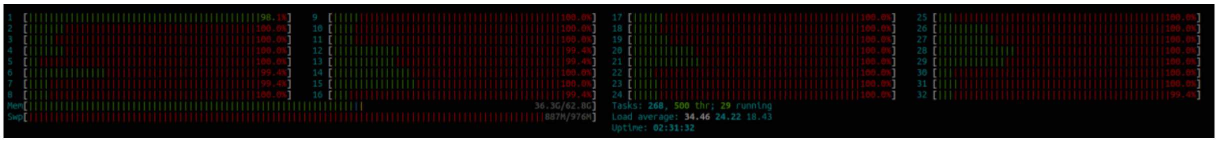

Figure 10. Use of 32 thread CPU to fuzz seriously

Even if only nine unique crashes out of 22.1 thousand have been detected, by debugging these nine crashes taken in a short period against the AddressSanitizer (ASan)[15] compiled binary, we can directly see that two "uniq crashes" recorded by AFL++ are in fact the same (thanks to backtrace information):

```
$ ../binaries/Fuzzy-afl-clang-fast-default default-fast-
default/crashes/id:000000,sig:11,src:000000,time:92,op:havoc,rep:8
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==34307==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address
0x000000000000 (pc 0x000000427897 bp 0x000000000001 sp 0x7fff3ff8adb0
T34307)
==34307==The signal is caused by a READ memory access.
==34307==Hint: address points to the zero page.
```

```
#0 0x427897 in GetElement
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:145:30
#1 0x427897 in LoRaMacConfirmQueueIsCmdActive
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:309:9
#2 0x4244df in ProcessRadioRxDone
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/main.c:1561:21
#3 0x4244df in main
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/main.c:2157:3
#4 0x7f771c8120b2 in __libc_start_main /build/glibc-eX1tMB/glibc-
2.31/csu/../csu/libc-start.c:308:16
#5 0x4034dd in _start
(/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/binaries/Fuzzy-afl-clang-fast-default+0x4034dd)

UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: SEGV
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:145:30 in GetElement
==34307==ABORTING

[…]

$ ../binaries/Fuzzy-afl-clang-fast-default default-fast-
default/crashes/id:000001,sig:11,src:000000,time:175,op:havoc,rep:8 1
↲
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==34343==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address
0x000000000000 (pc 0x000000427897 bp 0x000000000001 sp 0x7ffc05236f10
T34343)
==34343==The signal is caused by a READ memory access.
==34343==Hint: address points to the zero page.
#0 0x427897 in GetElement
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:145:30
#1 0x427897 in LoRaMacConfirmQueueIsCmdActive
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:309:9
#2 0x4244df in ProcessRadioRxDone
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/main.c:1561:21
#3 0x4244df in main
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/main.c:2157:3
#4 0x7fa22e1810b2 in __libc_start_main /build/glibc-eX1tMB/glibc-
```

```
2.31/csu/../csu/libc-start.c:308:16
#5 0x4034dd in _start
(/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/binaries/Fuzzy-afl-clang-fast-default+0x4034dd)

UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: SEGV
/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Fuzz/LoRaMacConfirmQueue.c:145:30 in GetElement
==34343==ABORTING
```

Here, only the address of the **__libc_start_main** function differs in the call stack, which can be irritating when one is dealing with many files. To resolve this small inconvenience, we processed the output of the ASan display and created a unique MD5 hash based on the call stack, excluding **__libc_start_main**, to get a unique crash trace.

After determining if a crash is unique given the unique ID hash, we classify the crash by its type as detected by ASan. A crash can be classified as either a leak type or a buffer overflow type, among others.

This helps us to focus not only on the most interesting bugs first, but also on the "quick wins."

To finish, we also need to know which payload does not crash a non-instrumented binary. This also helps us focus directly on the most interesting bugs. That step can simply be achieved using a GDB script that will run, show a backtrace, and quit the debugging process:

```
$ cat run.gdb
r
bt
quit
```

It can be run as follows:

```
$ gdb --batch --command=scripts/run.gdb --args binaries/Original-gcc
foutput/default-fast-
default/crashes/id:000000,sig:11,src:000000,time:92,op:havoc,rep:8 1
1testtest[Inferior 1 (process 35433) exited normally]
scripts/run.gdb:2: Error in sourced command file:
No stack.
```

In this context, we see that the crash is not triggered with a non-instrumented binary, so it is possible that this payload should be analyzed later.

The result from our classification engine is then recorded into an HTML report file as follows:

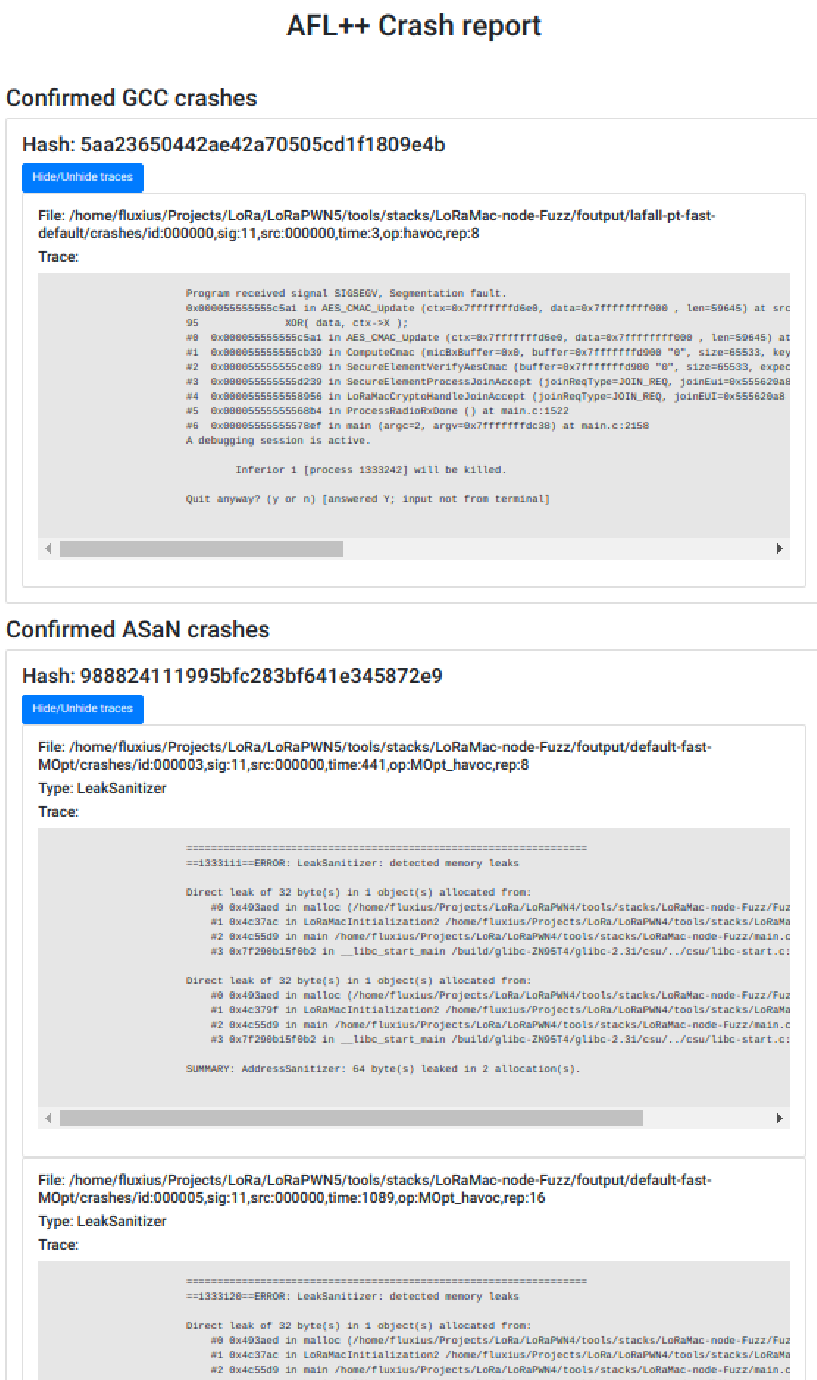

Figure 11. Results of the classification crash report when fuzzing before v4.5.1 of LoRaMac-node[16]

Payload minimalization

There are two ways to minimalize the payload. One is through corpus minimalization and another is by test case minimalization.

Corpus minimalization can be performed with the **afl-cmin** tool, which will find the smallest subset of files that will perform as much coverage as possible. The test case minimalization offered by **afl-tmin** offers a way to remove much of the data while keeping the same state of covered path or crash.

This takes time, but some tools are also available to speed up the process:

- https://github.com/googleprojectzero/halfempty
- https://github.com/MarkusTeufelberger/afl-ddmin-mod
- https://github.com/ilsani/afl-pytmin

We will consider integrating these tools into the architecture in the future.

Emulation

Fuzzing source code with AFL++ is the most scalable technique when the payloads are generated to pass as much code path as they can and are also reduced to the minimum size. But as we saw earlier, the code is compiled to a different architecture than x86-64, as well as with a specific cross compiler containing specific options. Therefore, if we try to prove the vulnerability by exploiting it, more time will be wasted adapting the exploit to the right architecture.

Some firmware can also be closed-source, so we need different methods other than static analysis to continue automatic bug finding.

Introducing stubs during debugging with GDB Python scripts or using Frida[17] on a few architectures supported by the tool is one method out of many that exist. Emulating with multiplatform engines such as Unicorn[18] or Qiling[19] is another.

For this article, we have decided to demonstrate the use of the Qiling framework, which is a valuable tool used to quickly develop proof-of-concept emulators for multiple types of architectures.

Building a LoRaMAC-node stack for a target

To demonstrate the tool in a straightforward way and with symbols, we chose the LoRaMAC-node project, which is open-source but compiled in ARM and mostly supported by the following platforms:

- NAMote72
- NucleoLxxx
- SKiM880B, SKiM980A, SKiM881AXL

- SAMR34

To begin, we compiled this stack for the NucleoL476 platform with a LR1110MB1DIS MBED shield (since it is the supported platform for this project):

```
$ cmake -DCMAKE_BUILD_TYPE=Release \
        -DTOOLCHAIN_PREFIX="/usr/bin/" \
        -DCMAKE_TOOLCHAIN_FILE="../cmake/toolchain-arm-none-
eabi.cmake" \
        -DAPPLICATION="LoRaMac" \
        -DSUB_PROJECT="periodic-uplink-lpp" \
        -DCLASSB_ENABLED="ON" \
        -DACTIVE_REGION="LORAMAC_REGION_EU868" \
        -DREGION_EU868="ON" \
        -DREGION_US915="OFF" \
        -DREGION_CN779="OFF" \
        -DREGION_EU433="OFF" \
        -DREGION_AU915="OFF" \
        -DREGION_AS923="OFF" \
        -DREGION_CN470="OFF" \
        -DREGION_KR920="OFF" \
        -DREGION_IN865="OFF" \
        -DREGION_RU864="OFF" \
        -DBOARD="NucleoL476" \
        -DMBED_RADIO_SHIELD="LR1110MB1XXS" \
        -DSECURE_ELEMENT="LR1110_SE" \
        -DSECURE_ELEMENT_PRE_PROVISIONED="ON" \
        -DUSE_RADIO_DEBUG="ON" ..
```

So, we got a binary file that looks as follows:

```
$ file LoRaMac-periodic-uplink-lpp*
LoRaMac-periodic-uplink-lpp:     ELF 32-bit LSB executable, ARM, EABI5
version 1 (SYSV), statically linked, with debug_info, not stripped
LoRaMac-periodic-uplink-lpp.bin: data
LoRaMac-periodic-uplink-lpp.hex: ASCII text, with CRLF line
terminators
```

The good thing about building this way is that we also have an ELF file that directly provides us with the entry point of our binary with section details. This could help us with the emulation part.

First run with Qiling

Qiling supports this architecture, as well as many others:

- X86
- X86_64
- Arm
- Arm64
- MIPS (only MSB for from now)
- 8086

This framework also provides many examples to run executables for many file formats:

- PE
- MachO
- ELF
- COM
- MBR

The Qiling documentation provides many examples and shows how to fuzz a complete binary using exotic architectures like those in routers. [20] Doing the same, we adapted the provided lines in the documentation with our own binary. The results are as follows:

```
ql = Qiling([“LoRaMac-periodic-uplink-lpp”], ".") # arg1=binary path,
arg2=rootfs
ql.run()
```

But running the binary directly with the few lines is not enough. Indeed, we can see that our binary crashes after some emulated code:

```
$ python3
emulate_demo.py

[x]

[x]r0: 0x20000000
[x]r1: 0xe000ed00
[x]r2: 0x20003064
[x]r3: 0x20003064
[x]r4: 0x0
[x]r5: 0x0
[x]r6: 0x0
[x]r7: 0x0
```

```
[x]r8: 0x0
[x]r9: 0x0
[x]r10: 0x0
[x]r11: 0x0
[x]r12: 0x0
[x]sp: 0x20018000
[x]lr: 0x800bfa3
[x]pc: 0x800bfc8
[x]cpsr: 0x600001f3
[x]c1_c0_2: 0xf00000
[x]c13_c0_3: 0x0
[x]fpexc: 0x40000000
[x]

[x]PC = 0x800bfc8
[x] (/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-
node-Emulate/LoRaMac-periodic-uplink-lpp+0x800bfc8)
[=][+] Start      End        Perm.  Path
[=][+] 08000000 - 08014000 - r-
x    /home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-
node-Emulate/LoRaMac-periodic-uplink-lpp
(/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Emulate/LoRaMac-periodic-uplink-lpp)
[=][+] 20000000 - 20004000 -
rw-    /home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-
node-Emulate/LoRaMac-periodic-uplink-lpp
(/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Emulate/LoRaMac-periodic-uplink-lpp)
[=][+] 20004000 - 20006000 - rwx    [hook_mem]
[=][+] 7ff0d000 - 7ff3d000 - rwx    [stack]
[=][+] ffff0000 - ffff1000 - rwx    [arm_tls]
[x]['0xf', '0x49', '0xd1', '0xf8', '0x88', '0x30', '0x43', '0xf4']
[=]

[=]0x0800bfc8
{/home/fluxius/Projects/LoRa/LoRaPWN_tool/tools/stacks/LoRaMac-node-
Emulate/LoRaMac-periodic-uplink-lpp + 0x00bfc8}   0f 49 d1 f8 88 30 43
f4 70 03 c1 f8 88 30 0d 4b 1a 68 00 20 42 f0 01 02 1a 60 98 60 1a 68
22 f0 a8 52 22 f4 10 22 1a 60 4f f4 80 52 da 60 1a 68 22 f4 80 22 1a
60 98 61 4f f0 00 63 8b 60 70 47 ldr r1, [pc, #0x3c]
> ldr.w r3, [r1, #0x88]
> orr r3, r3, #0xf00000
> str.w r3, [r1, #0x88]
> ldr r3, [pc, #0x34]
> ldr r2, [r3]
> movs r0, #0
> orr r2, r2, #1
> str r2, [r3]
```

```
> str r0, [r3, #8]
> ldr r2, [r3]
> bic r2, r2, #0x15000000
> bic r2, r2, #0x90000
> str r2, [r3]
> mov.w r2, #0x1000
> str r2, [r3, #0xc]
> ldr r2, [r3]
> bic r2, r2, #0x40000
> str r2, [r3]
> str r0, [r3, #0x18]
> mov.w r3, #0x8000000
> str r3, [r1, #8]
> bx lr
Traceback (most recent call last):
  File "emulate_demo.py", line 4, in <module>
    ql.run()
  File "/home/fluxius/.local/lib/python3.8/site-
packages/qiling/core.py", line 756, in run
    self.os.run()
  File "/home/fluxius/.local/lib/python3.8/site-
packages/qiling/os/linux/linux.py", line 118, in run
    self.ql.emu_start(self.ql.loader.elf_entry, self.exit_point,
self.ql.timeout, self.ql.count)
  File "/home/fluxius/.local/lib/python3.8/site-
packages/qiling/core.py", line 897, in emu_start
    self.uc.emu_start(begin, end, timeout, count)
  File "/usr/local/lib/python3.8/dist-packages/unicorn/unicorn.py",
line 318, in emu_start
    raise UcError(status)
unicorn.unicorn.UcError: Invalid memory read (UC_ERR_READ_UNMAPPED)
```

Patching the execution

To solve the issue, we need to dynamically allocate memory by adding the following function:

```
def memory_fix(ql, access, addr, size, value):
        if mem_map_force is True:
            ql.log.debug("[_] Mapping "+str(size)+" bytes at
"+hex(addr)+" | access: "+ str(access)+" | value: "+ str(value))
            ql.mem.map(addr//4096*4096, 4096)
            ql.mem.write(addr, struct.pack(">I",value)) # memory
packing is OS dependant
        else:
```

```
            print(("Auto-Memmap disabled for this address"))
        return
```

After this, we use an unmapped memory hook to call our function each time the problem "reading or writing to an unmapped memory" happens:

```
ql.hook_mem_unmapped(memory_fix)
```

We also make use of trace function with the power of the Capstone engine[21] to disassemble all instruction if we want to, as well as disable the initial debugging output to have something custom:

```
[...]
from capstone import *
from binascii import hexlify
from capstone.arm import *
[...]
if enable_trace is False:
        outputd = "off"
        enable_trace = True

    ql = Qiling([binary_file], ".",
                output=outputd,
            stdout=1 if enable_trace else None,
                stderr=1 if enable_trace else None,
                console = True if enable_trace else False)
md = Cs(CS_ARCH_ARM, CS_MODE_THUMB)
count = [0]
[...]
def trace_cb(ql, address, size, count):
        dis = disasm(count, ql, address, size)
        if dis is not None:
            ql.log.debug(dis)
        count[0] += 1
if enable_trace:
        ql.hook_code(trace_cb, count)
```

By fixing the memory, our program runs like a charm — except that it runs like an infinite loop after the BLX on R3 at address 0x08010758:

```
[+]     00003F9F        08010758: 98
47                   blx        r3
[+]     [_] Mapping 1 bytes at 0x0 | access: 21 | value: 0
[+]     00003FA0        00000000: 00 00 00
00              movs        r0, r0
[+]     00003FA1        00000004: 00 00 00
```

```
00              movs        r0, r0
[+]     00003FA2        00000008: 00 00 00
00              movs        r0, r0
[+]     00003FA3        0000000C: 00 00 00
00              movs        r0, r0
[+]     00003FA4        00000010: 00 00 00
00              movs        r0, r0
[+]     00003FA5        00000014: 00 00 00
00              movs        r0, r0
[...]
```

Using Ghidra, we can clearly see at this address that a call to the *arm_set_fast_math* function is done, but the address is missing:

```
                     LAB_08010742                                    XREF[1]:     0801073
08010742 09 4e        ldr        r6,[->__do_global_dtors_aux_fini_array_entry]   = 20
08010744 09 4d        ldr        r5,[->__preinit_array_end]                      = 20
08010746 76 1b        sub        r6,r6,r5
08010748 01 f0 76 fc  bl         _init                                           int
0801074c b6 10        asr        r6,r6,#0x2
0801074e 06 d0        beq        LAB_0801075e
08010750 00 24        mov        r4,#0x0

                     LAB_08010752                                    XREF[1]:     0801075
08010752 01 34        add        r4,#0x1
08010754 55 f8 04 3b  ldr.w      r3,[r5],#0x4=>__preinit_array_end               = 08
                                                                                 = 08
08010758 98 47        blx        r3=>__arm_set_fast_math                         unde
                                                                                 unde
0801075a a6 42        cmp        r6,r4
0801075c f9 d1        bne        LAB_08010752
```

Figure 12. A missing address call

Based on Ghidra, however, the function clearly exists:

```
                     undefined __arm_set_fast_math()
                       assume LRset = 0x0
                       assume TMode = 0x1
     undefined         r0:1          <RETURN>
                     __arm_set_fast_math                             XREF[3]:     __libc_
                                                                                  __libc_
                                                                                  .debug_
08000188 f1 ee 10 3a   mrc        p10,0x7,r3,cr1,cr0,0x0
0800018c 43 f0 80 73   orr        r3,r3,#0x1000000
08000190 e1 ee 10 3a   vmsr       fpscr,r3
08000194 70 47         bx         lr
08000196 00            ??         00h
08000197 bf            ??         BFh
```

Figure 13. Existing function inside the binary

To resolve this, we made a quick fix with a new hook:

```
[...]
def fix_arm_set_fmath_addr(ql):
        ql.reg.r3 = 0x08000188
[...]
ql.hook_address(fix_arm_set_fmath_addr, 0x08010758)
```

But that was not the last problem in our journey. Indeed, many registers will require fixes to run the program properly:

```
[...]
[+]     00003F9F        08010758: 98
47                    blx        r3
[+]     00003FA0        08000188: f1 ee 10
3a              vmrs       r3, fpscr
[+]     [_] Mapping 1 bytes at 0x843bd54 | access: 21 | value: 0
[+]     00003FA1        0843BD54: 00 00 00
00              movs       r0, r0
[+]     00003FA2        0843BD58: 00 00 00
00              movs       r0, r0
 [...]
```

Although we do not go through all of these issues, we will talk about other problems that might come up with regard to platform-specific calls that could waste time. The following are examples:

- BoardInit()
- SecureElementInit()
- lr1110_radio_set_lora_sync_word()
- GpioWrite()
- TimerStart()

We can simply get rid of all these calls using a function that will patch all call instructions doing NOPs (a specific instruction that does nothing) proper to ARM. If there are issues, we can also use the Keystone engine[22] that could give the right operation code for the targeted instruction set, as seen here:

```
$ kstool thumb "nop"
nop = [ 00 bf ]
```

This results in the following patch:

```
[...]
nop_addresses = { #0x0800bf9e : b"\x00\xbf" * 2,
                  0x0800bfa2 : b"\x00\xbf" * 2,
                  #0x0800aaa2 : b"\x00\xbf" * 2,
                  0x08002b32 : b"\x00\xbf" * 2, # BoardInit()
                  #0x08002b36 : b"\x00\xbf" * 2, # BoardInitPeriph()
                  0x08005bc8 : b"\x00\xbf", # bypass
LORAMAC_STATUS_REGION_NOT_SUPPORTED condition
                  0x08005e3a : b"\x00\xbf", # RadioInit()
                  #0x08005e40 : b"\x00\xbf" * 2, #SecureElementInit()
                  0x0800a72e : b"\x00\xbf" * 18, # lr1110_* in
SecureElementInit()
                  0x0800918a : b"\x00\xbf" * 3,  # RadioStandby() +
result in r0
                  0x08009192 : b"\x00\xbf" * 2, #
lr110_system_get_random_number
                  0x08009742 : b"\x00\xbf" * 2, # RadioSetModem()
                  0x0800a066 : b"\x00\xbf" * 2, #
lr1110_radio_set_lora_sync_word()->lr1110_hal_write()
                  0x08005e7e : b"\x00\xbf", # RadioSleep()
                  0x08000cec : b"\x00\xbf", # Skip branch
                  0x08000d8a : b"\x00\xbf", # force
LORAMAC_HANDER_SUCCESS
                  0x0800522c : b"\x00\xbf" * 2, #
BoardCriticalSection()
                  0x0800523c : b"\x00\xbf" * 2, #
BoardCriticalSectionEnd()
                  0x08005242 : b"\x00\xbf", # bypassing Event check
                  0x08005306 : b"\x00\xbf", # bypassing Event check 2
                  0x08005368 : b"\x00\xbf", # RadioSleep()
                  0x080087ae : b"\x00\xbf" * 2, #
SecureElementProcessJoinAccept()
                  0x080087b4 : b"\x00\xbf", # Force
SECURE_ELEMENT_SUCCESS
                  0x080056dc : b"\x00\xbf", # Force
LORAMAC_CRYPTO_SUCCESS
                  0x08002a54 : b"\x00\xbf", # OnRXData->GpioWrite()
                  0x08002a5e : b"\x00\xbf", # OnRXData-->TimerStart()
                }
[...]
def skip_it(ql, list_instru): # patch broken instructions
        for instru, rcode in list_instru.items():
            ql.patch(instru, rcode)
```

```
skip_it(ql, nop_addresses)
```

After all the fixes, we can run the program without a problem and finish its execution:

```
[...]
[+]00013B4E08005622: 9a 07                        lsls       r2, r3,
#0x1e
[+]00013B4F08005624: 08
d5                   bpl        #0x8005638
[+]00013B5008005638: 94 f8 8c 34                  ldrb.w     r3, [r4,
#0x48c]
[+]00013B510800563C: 02 2b                        cmp        r3,
#2
[+]00013B520800563E: 01
d1                   bne        #0x8005644
[+]00013B5308005644: 29 b0                        add        sp,
#0xa4
[+]00013B5408005646: bd e8 f0 8f                  pop.w      {r4, r5, r6,
r7, r8, sb, sl, fp, pc}
```

Reimplementing some functions

Notably, reading such instructions can be exhausting. This is why user-friendly debugging methods are always welcome. Indeed, we can see that the binary also makes use of some *printf()* functions as follows:

```
                                                                    = 20
08002b98 fe f7 7c f8    bl      LmHandlerInit                      LmHa
08002b9c 04 46          mov     r4,r0
08002b9e 18 bl          cbz     r0,LAB_08002ba8
08002ba0 42 48          ldr     r0=>s_LoRaMac_wasn't_properly_initiali_08012ca... = "L
                                                                    = 08
08002ba2 0d f0 4d fe    bl      printf                             int
```

Figure 14. *printf()* function in the binary

We can therefore use these calls to make some hooks to a homemade function in Python that will take the arguments past the function and simply print everything as it should be:

```
def hijack_printf(ql):
        new_str = ""
```

```
        fmt = ql.mem.string(ql.os.function_arg[0])
        matches = re.findall("\%\w+", fmt)
        count = 0
        for sp in fmt.split("%"):
            if count == 0:
                new_str += sp
            else:
                if matches[count-1] == "%s":
                    new_str +=
ql.mem.string(ql.os.function_arg[count])+ sp[1:]
                elif matches[count-1] == "%d" or matches[count-1] ==
"%i":
                    new_str += "%d" % int(ql.os.function_arg[count])+
sp[1:]
            count += 1
        print (new_str)
```

We can then have these beautiful prints when running the binary:

```
$ python3 emul_LoRaMacNode.py -b LoRaMac-periodic-uplink-lpp -
v

###### =================================== ######


Application name   : periodic-uplink-lpp

Application version: 1.2.0

GitHub base version: 4.5.0


###### =================================== ######
```

But this is not finished yet. We also need to emulate the binary and input packets to parse there, and we have not even made use of the parser yet.

Parsing LoRaWAN packets

To parse our packet, we make use of a pipe (as used in the fuzzing demonstration with AFL that we discuss in later sections):

```
[...]
class MyPipe():
    def __init__(self):
        self.buf = b''

    def write(self, s):
        self.buf += s

    def read(self, size):
        if size <= len(self.buf):
            ret = self.buf[: size]
            self.buf = self.buf[size:]
        else:
            ret = self.buf
            self.buf = ''
        return ret

    def fileno(self):
        return 0

    def show(self):
        pass

    def clear(self):
        pass

    def flush(self):
        pass

    def close(self):
        self.outpipe.close()

    def fstat(self):
        return stdin_fstat
[...]
def main(binary_file, enable_trace=False, enable_verbose=False,
message_bytes=b"", input_file=None, output_file=None):
    global mem_map_force
    global inject_addr

    stdin = MyPipe()

    # for unicorn_afl
    outputd = "debug"
    if enable_trace is False:
        outputd = "off"
        enable_trace = True
    # end
```

```
    ql = Qiling([binary_file], ".",
                output=outputd,
                stdin=stdin,
                stdout=1 if enable_trace else None,
                stderr=1 if enable_trace else None,
                console = True if enable_trace else False)

    md = Cs(CS_ARCH_ARM, CS_MODE_THUMB)
    count = [0]
[...]
```

This allows us to provide an input packet with our command line, but we also need to use the parser, inject the message, and process it. To do so, we will use a new hook that will jump to the parser after the initialization of the binary to get a stable context:

```
[...]
   def jump2parser(ql):
        global mem_map_force
        mem_map_force = False # Don't force map anything from now
        inject_msg(message_bytes)
        # Jump to the parser
        ql.reg.pc = 0x08005225 # thump jump to parser
[...]
ql.hook_address(jump2parser, 0x08002bb6)
[...]
```

Adding other debugging hooks allows us to parse a join-accept type packet, resulting in the following:

```
$ python3 emul_LoRaMacNode2.py -b LoRaMac-periodic-uplink-lpp -v -s
[JOINT ACCEPT PKT]

###### ===================================== ######


Application name   : periodic-uplink-lpp

Application version: 1.2.0

GitHub base version: 4.5.0
```

```
###### ===================================== ######


Mapping payload at:  0x1000
Parsing case:  FRAME_TYPE_JOIN_ACCEPT
Parsing case:  FRAME_TYPE_DATA_UNCONFIRMED_DOWN
```

This is perfect for us if we find some bugs that we want to confirm as exploitable vulnerabilities. We can make an exploit without tweaking the payload too much, depending on the context (mitigations and address space).

These are not the only features available in Qiling, however. In fact, we can also use Qiling with a patched Unicorn Engine stub with AFL to do some fuzzing tests. But before delving into fuzzing, let us first optimize the execution to speed up the fuzzing process also.

Optimize execution speed

Qiling has a notable feature called snapshot[23] that can speed up the execution process. To make use of it, we can snapshot the execution of the binary when we want to jump into our parser with the *save()* function of Qiling, as follows:

```
def jump2parser(ql):
        global mem_map_force
        mem_map_force = False # Don't force map anything from now
        inject_msg(message_bytes)
        # Jump to the parser
        ql.save(reg=True, cpu_context=True, snapshot="snapshot.bin")
        ql.reg.pc = 0x08005225 # thump jump to parser
```

After one run, a snapshot should be written in the current directory:

```
$ ls -lh snapshot.bin
-rw-rw-r-- 1 fluxius fluxius 340K févr. 26 09:45 snapshot.bin
```

For the next runs, we can restore the snapshot, disable the unmapped memory hooks, and directly run at the packet parser's address and define an end to the execution (as seen in the following). Then, we can start fuzzing the proper way with Qiling.

```
 [...]
md = Cs(CS_ARCH_ARM, CS_MODE_THUMB)
count = [0]

ql.restore(snapshot="snapshot.bin")
[…]
#ql.hook_mem_unmapped(memory_fix)
[…]
#ql.run()
ql.run(begin=0x08005225, end=0x800563e)
[...]
```

Fuzzing with Qiling

Qiling brings the UnicornAFL[24] feature to the game, so we not only use the framework to emulate, but also fuzz an emulated binary of a different platform.

Using the feature is a straightforward matter. First, we need to load a patched Unicorn version, define a function to start AFL, and finally, use a hook at the address that should start the fuzzing process:

```
import unicornafl
unicornafl.monkeypatch()
[…]
    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:
            #print("Starting afl_fuzz().")
            if not _ql.uc.afl_fuzz(input_file=input_file,
                        place_input_callback=place_input_callback,
                        exits=[ql.os.exit_point]):
                print("Ran once without AFL attached.")
                os._exit(0)  # that's a looot faster than tidying up.
        except unicornafl.UcAflError as ex:
            if ex != unicornafl.UC_AFL_RET_CALLED_TWICE:
                raise
     […]
    # Fuzzing hook
    ql.hook_address(start_afl, 0x800522c)
    #ql.run()
    ql.run(begin=0x08005225, end=0x800563e)
```

To finish, we write a starting script to launch all the things in an easy manner:

```
#!/bin/bash

if [ ! -d ./AFLplusplus ]; then
  git clone https://github.com/AFLplusplus/AFLplusplus.git
  cd AFLplusplus
  make
  cd ./unicorn_mode
  ./build_unicorn_support.sh
  cd ../../
fi

AFL_AUTORESUME=1 AFL_PATH="$(realpath ./AFLplusplus)"
PATH="$AFL_PATH:$PATH" afl-fuzz -t <some fuzzy values> -i afl_inputs -
o afl_outputs -U -- python3 emul_LoRaMacNode.py -b LoRaMac-periodic-
uplink-lpp --fuzz_input @@
```

But at the end, even with optimization, we face the limitations of the framework in Python 3, leading with just 1.54 executions per second on an i7 vPro 10th Gen computer:

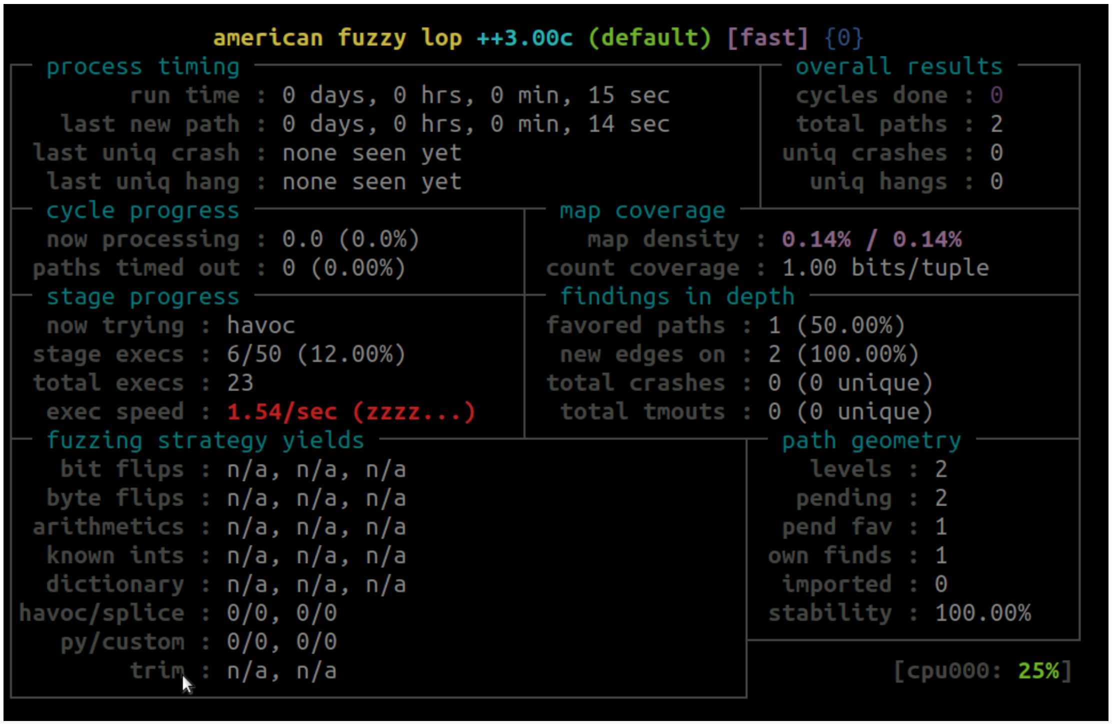

Figure 15. AFLUnicorn with Qiling

Unicorn Engine emulation in C would be a better candidate for this task after doing the quick proof-of-concept with Qiling in Python. Nevertheless, Qiling can be considered for fuzzing smaller code paths, or by making more optimizations than what is shown in this example.

Emulating and fuzzing with Ghidra

We have seen architecture supported by Unicorn and Qiling, which gives us the ability to emulate and fuzz ARM architecture. But when it comes to emulating and fuzzing gateways, the architecture that is often encountered is MIPS MSB, which is not yet handled by Unicorn and Qiling. As a result, we opted for Ghidra for these architectures.

It is also possible to use Ghidra with official processors as an alternative.[25] For example, users can perform emulation with extended processors like Xtensa[26] on Espressif chips.

It should be noted that to emulate the parsing function of a LoRaWAN gateway, the parsing function must be enabled to act in standalone mode. In LoRaWAN, it is rare to find a gateway parsing the packet from an end-node, but this situation can happen if the gateway is put in standalone mode and it is able to parse packets in this mode.

To emulate the parsing function that will be working in MIPS MSB architecture, we can make use of Ghidra by creating either a Python or a Java module.

For this section, we have quickly adapted the script from a very detailed article by John Toterhi about Ghidra PCode emulation in X86.[27] First, we import modules like the emulation helper, as well as the module that can help us give pointers to some symbol names. Then we define helpers that will simplify getting the list of registers and addresses of symbols:

```
# adapted code from John Toterhi's article
 from ghidra.app.emulator import EmulatorHelper
from ghidra.program.model.symbol import SymbolUtilities

# == Helper functions
======================================================
def getAddress(offset):
    return
currentProgram.getAddressFactory().getDefaultAddressSpace().getAddress
(offset)


def getSymbolAddress(symbolName):
    symbol = SymbolUtilities.getLabelOrFunctionSymbol(currentProgram,
symbolName, None)
    if (symbol != None):
        return symbol.getAddress()
    else:
        raise Exception("Failed to locate label:
{}".format(symbolName))
```

```
def getProgramRegisterList(currentProgram):
    pc = currentProgram.getProgramContext()
    return pc.registers
```

We will then create a *main()* function that will, once called, get the address of the *LoRaMacParserData()* function that will be called by filling PC registers with its address:

```
def main():
    CONTROLLED_RETURN_OFFSET = 0
    mainFunctionEntry = getSymbolAddress(" LoRaMacParserData ")
    emuHelper = EmulatorHelper(currentProgram)
    # Set controlled return location so we can identify return from
emulated function
    controlledReturnAddr = getAddress(CONTROLLED_RETURN_OFFSET)
    # Set initial PC
    mainFunctionEntryLong = int("0x{}".format(mainFunctionEntry), 16)
    emuHelper.writeRegister(emuHelper.getPCRegister(),
mainFunctionEntryLong)
```

Afterward, we finish our *main()* function that will make use of a monitor to single-step the emulated instruction one by one, until we reach the 0x0 invalid address:

```
    registers = getProgramRegisterList(currentProgram)


    # Here's a list of all the registers we want printed after each
    # instruction. Modify this as you see fit, based on your
architecture.
    reg_filter = [
        "zero", "at", "v0", "v1", "a0",
      "a1", "a2", "a3", "t0", "t1",
      "t2", "t3", "t4", "t5", "t6",
        "t7", "s0", "s1", "s2", "s3",
      "s4", "s5", "s6", "s7", "t8",
      "t9", "k0", "k1", "gp", "sp",
      "s8", "ra", "pc",
    ]


    print("Emulation starting at 0x{}".format(mainFunctionEntry))
    while monitor.isCancelled() is False:
```

```python
        # Check the current address in the program counter, if it's
        # zero (our `CONTROLLED_RETURN_OFFSET` value) stop emulation.
        # Set this to whatever end target you want.
        executionAddress = emuHelper.getExecutionAddress()
        if (executionAddress == controlledReturnAddr):
            print("Emulation complete.")
            return


        # Print current instruction and the registers we care about
        print("Address: 0x{} ({})".format(executionAddress,
getInstructionAt(executionAddress)))
        for reg in reg_filter:
            reg_value = emuHelper.readRegister(reg)
            print("  {} = {:#018x}".format(reg, reg_value))


        # single step emulation
        success = emuHelper.step(monitor)
        if (success == False):
            lastError = emuHelper.getLastError()
            printerr("Emulation Error: '{}'".format(lastError))
            return


    # Cleanup resources and release hold on currentProgram
    emuHelper.dispose()


# == Invoke main
===========================================================
main()
```

By running this script, we get the first result as follows:

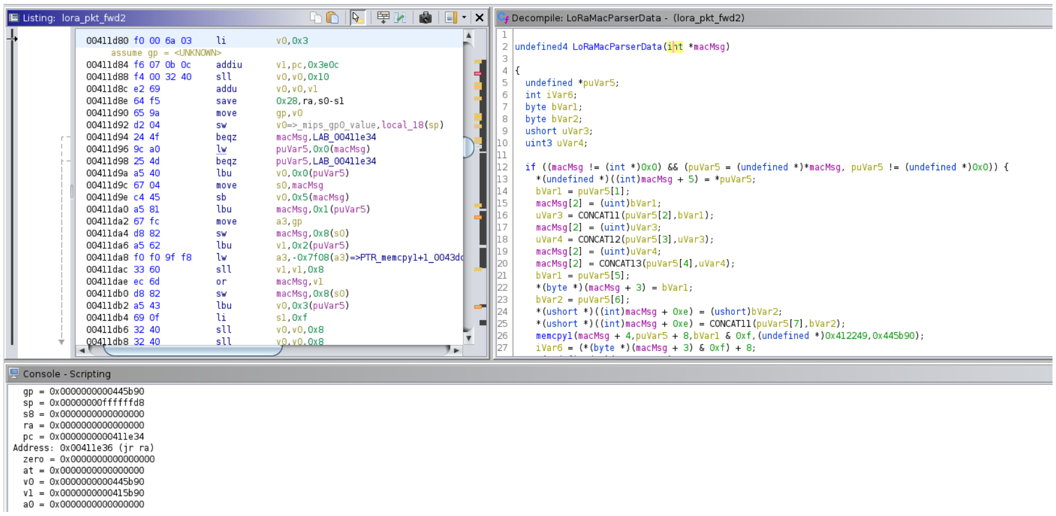

Figure 16. Emulation with Ghidra

Some memory contexts will be required to run the function properly or to force cases (exactly like with Qiling). We will then have to make use of **emuHelper.write*** helpers to set up registers and memory with a proper state.

To perform the fuzzing, we look to an informative project of Flavian Dola from Airbus.[28] The project was published running a trampoline program with AFL++ to forward input to the target, as seen in Figure 17.

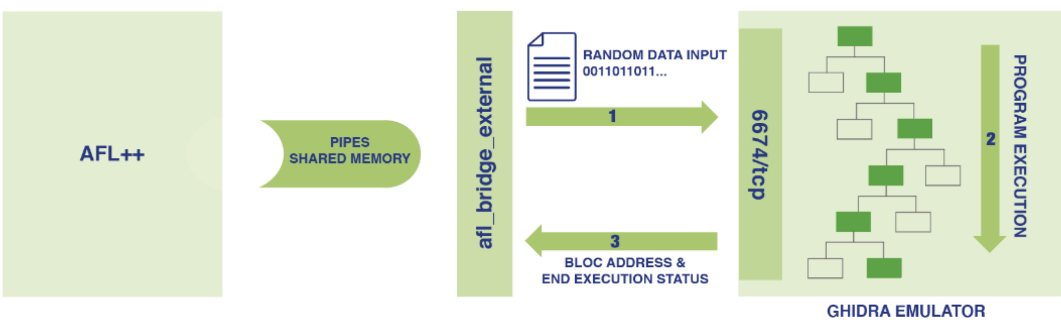

Figure 17. AFL Ghidra emulator PoC architecture

To go further, we encourage the reader to take a look at the documentation of this project where Airbus also gives examples of Xtenza and PPC targets.[29]

Conclusion and Recommendations

It is important to trust the LoRaWAN protocol stack implementation, and this level of trust can only be achieved by constantly testing it against memory corruptions and logical bugs. To do so, it is recommended to first choose a protocol stack that was approved by the community and also tested by security researchers. Afterward, it is important to invest in resources and spend time fuzzing environments to check if the libraries used are resistant to most of the test cases scenarios, as shown in the previous sections using different techniques.

In our report, we covered only targeted parser fuzzing, but complete fuzzing scenarios must be also integrated into the audits and stress tests of the whole application to certify the robustness of the stack.

The image here shows an example of how fuzzing tests can be integrated in the battery of tests usually done before releasing the product.

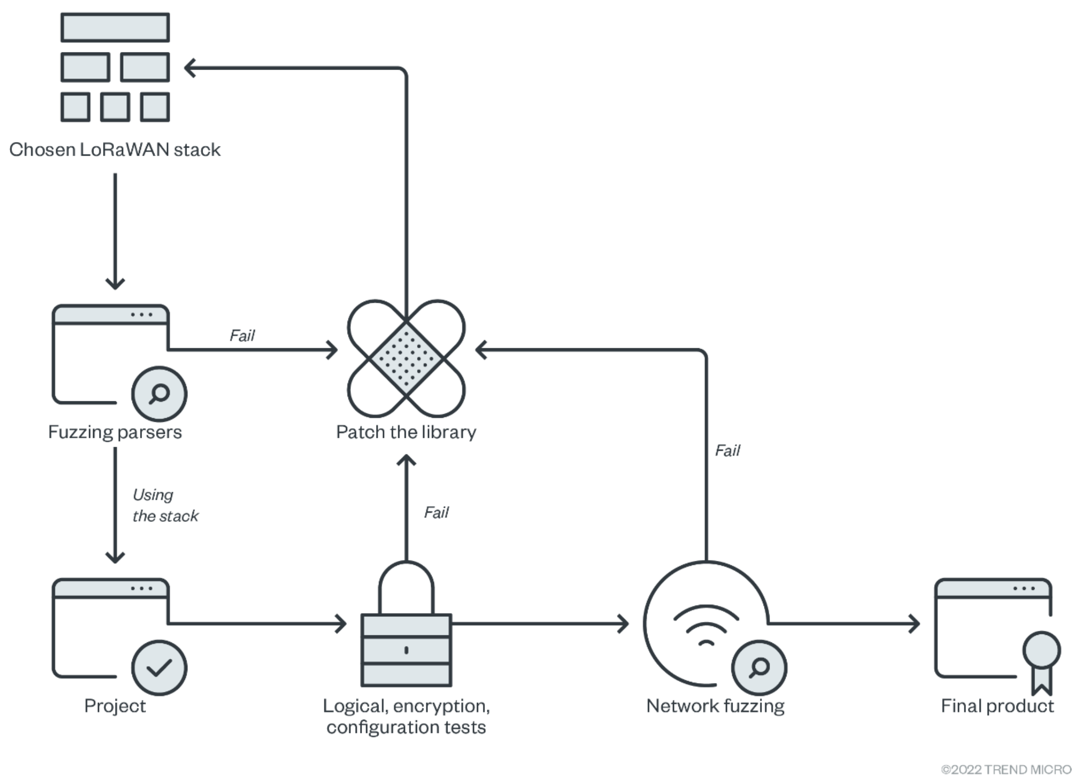

Figure 18. Fuzzing integrated into the battery of tests

By imagining ourselves with an attacker's mindset, we are able to understand possible security issues and flaws and find additional attack vectors that were not covered by our previous research. Although we have already highlighted the complexity of these security issues in the previous sections of this technical brief, we also want to mention the complexity of the exploitation itself.

Ultimately, the attacker would have to know precisely what the target is, how it was compiled, or (by chance) get a dump of the firmware. Nevertheless, despite this high level of complexity, this class of bugs must be taken seriously if we want to guarantee solid security inside industrial factories or smart city environments using LoRaWAN technology.

References

[1]Sébastien Dudek. (Jan. 26, 2021). *Trend Micro Security Intelligence Blog.* “Low Powered and High Risk: Possible Attacks on LoRaWAN Devices.” Accessed on Jan. 6, 2022, at https://www.trendmicro.com/en_us/research/21/a/Low-Powered-but-High-Risk-Evaluating-Possible-Attacks-on-LoRaWAN-Devices.html.

[2]Sébastien Dudek. (Feb. 19, 2021). *Trend Micro Security Intelligence Blog.* “Gauging LoRaWAN Communication Security with LoraPWN.” Accessed on Jan. 6, 2022, at https://www.trendmicro.com/en_us/research/21/b/gauging-lorawan-communication-security-with-lorapwn.html.

[3]Sébastien Dudek. (March 30, 2021). *Trend Micro Security Intelligence Blog.* “Protecting LoRaWAN Hardware from Attacks in the Wild.” Accessed on Jan. 6, 2022, at https://www.trendmicro.com/en_us/research/21/c/protecting-lorawan-hardware-from-attacks-in-the-wild.html.

[4]Sébastien Dudek. (Feb. 5, 2021). *YouTube.* “LoRaPWNing: Practical radio attacks on LoRaWAN.” Accessed on Jan. 6, 2022, at https://www.youtube.com/watch?v=z-jSiR3-xW4.

[5]American Fuzzy Lop plus plus. (Jul. 19, 2021). *GitHub.* “AFL++.” Accessed on Jan. 6, 2022, at https://github.com/AFLplusplus/AFLplusplus.

[6]Qiling.io. (Dec. 29, 2021). *GitHub.* “Qiling Advanced Binary Emulation Framework.” Accessed on Jan. 6, 2022, at https://github.com/qilingframework/qiling.

[7]Semtech. (May 31, 2021). *GitHub.* “LoRaMac-node.” Accessed on Jan. 6, 2022, at https://github.com/Lora-net/LoRaMac-node.

[8]LoRa Basics™ Station. (June 6, 2020). *GitHub.* “Basics Station.” Accessed on Jan. 6, 2022, at https://github.com/lorabasics/basicstation.

[9]The Things Network. (n.d.). *The Things Network*. “The Things Network.” Accessed on Jan. 6, 2022, at https://www.thethingsnetwork.org/.

[10]The Things Network. (Feb. 5, 2021). *YouTube*. “The Things Network Upgrades to The Things Stack V3.” Accessed on Jan. 6, 2022, at https://www.youtube.com/watch?v=TtDE_5JNAGs.

[11]Semtech. (n.d.). *GitHub*. “packet forwarder of Dragino LG308.” Accessed on Jan. 6, 2022, at https://github.com/dragino/dragino-packages/blob/lg02/lora-gateway/src/lora_pkt_fwd/src/lora_pkt_fwd.c.

[12]Sébastien Dudek. (Feb. 19, 2021). *Trend Micro Security Intelligence Blog.* “Gauging LoRaWAN Communication Security with LoraPWN.” Accessed on Jan. 6, 2022, at https://www.trendmicro.com/en_us/research/21/b/gauging-lorawan-communication-security-with-lorapwn.html.

[13]Sébastien Dudek. (n.d.). *Scapy*. “loraphy2wan Scapy Layer.” Accessed on Jan. 6, 2022, at https://scapy.readthedocs.io/en/latest/api/scapy.contrib.loraphy2wan.html.

[14]Jonathan Salwan. (April 27, 2020). *GitHub*. "Triton Framework." Accessed on Jan. 6, 2022, at https://github.com/JonathanSalwan/Triton.

[15]LLVM. (n.d.) *Clang LLVM*. "Clang 13 Documentation AddressSanitizer." Accessed on Jan. 6, 2022, at https://clang.llvm.org/docs/AddressSanitizer.html.

[16]Lora-net. (April 16, 2020). *GitHub*. "LoRaMAC-node commit patching CVE-2020-11068." Accessed on Jan. 6, 2022, at https://github.com/Lora-net/LoRaMac-node/commit/e3063a91daa7ad8a687223efa63079f0c24568e4.

[17]Frida. (n.d.). *Frida*. "Frida." Accessed on Jan. 6, 2022, at https://frida.re/.

[18]Unicorn. (n.d.). *Unicorn Engine.* "Unicorn Engine." Accessed on Jan. 6, 2022, at https://www.unicorn-engine.org/.

[19]Qiling. (n.d.). *Qiling.io*. "Qiling Framework." Accessed on Jan. 6, 2022, at https://qiling.io/.

[20]Qiling Framework Documentation. (n.d.). *Qiling.io*. "Getting started." Accessed on Jan. 6, 2022, at https://docs.qiling.io/en/latest/howto/.

[21]Capstone. (n.d.). *Capstone Engine*. "Capstone The Ultimate Disassembler." Accessed on Jan. 6, 2022, at https://www.capstone-engine.org/.

[22]Keystone. (n.d.). *Keystone Engine*. "Keystone Engine The Ultimate Assembler." Accessed on Jan. 6, 2022, at https://www.keystone-engine.org/.

[23]Qiling Framework Documentation. (n.d.). *Qiling.io*. "Qiling Snapshot." Accessed on Jan. 6, 2022, at https://docs.qiling.io/en/latest/snapshot/.

[24]UnicornAFL. (n.d.). *GitHub*. "AFL bindings for Unicorn-Engine." Accessed on Jan. 6, 2022, at https://github.com/AFLplusplus/unicornafl.

[25]National Security Agency. (n.d.). *GitHub*. "Ghidra Processors." Accessed on Jan. 6, 2022, at https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Processors.

[26]Ebiroll. (n.d.). *GitHub*. "Ghidra Xtenza Extension." Accessed on Jan. 6, 2022, at https://github.com/Ebiroll/ghidra-xtensa.

[27]John Toterhi. (Jan. 27, 2020). *Medium*. "Emulating Ghidra's PCode: Why/How." Accessed on Jan. 6, 2022, at https://medium.com/@cetfor/emulating-ghidras-pcode-why-how-dd736d22dfb.

[28]Flavian Dola. (April 27, 2021). *Airbus*. "Fuzzing exotic arch with AFL using ghidra emulator." Accessed on Jan. 6, 2022, at https://airbus-cyber-security.com/fuzzing-exotic-arch-with-afl-using-ghidra-emulator/.

[29]Flavian Dola. (April 27, 2021). *Airbus*. "Fuzzing exotic arch with AFL using ghidra emulator." Accessed on Jan. 6, 2022, at https://airbus-cyber-security.com/fuzzing-exotic-arch-with-afl-using-ghidra-emulator/.