# Gecko Bootloader Vulnerability Research

HardwearIO

Sami Babigeon, Benoît Forgette - Whatever Analysis

November 1st, 2023

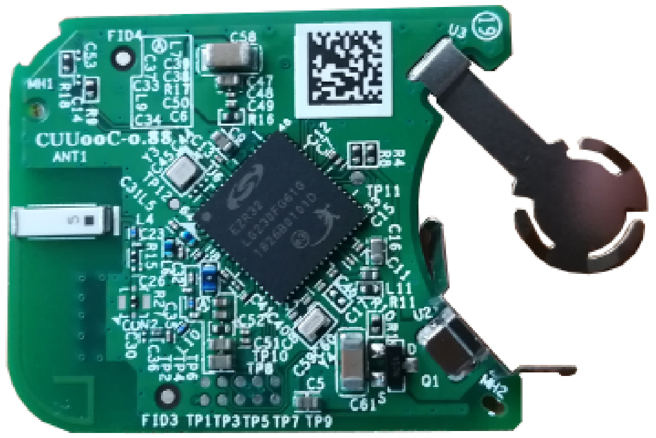
Quarkslab

## Who are we?

**Sami Babigeon**
Junior Security Researcher
topic (Fixing my computer)

**Benoît Forgette**
Software and hardware Security Researcher
topic (Hardware/Android)

## Goal of the Research

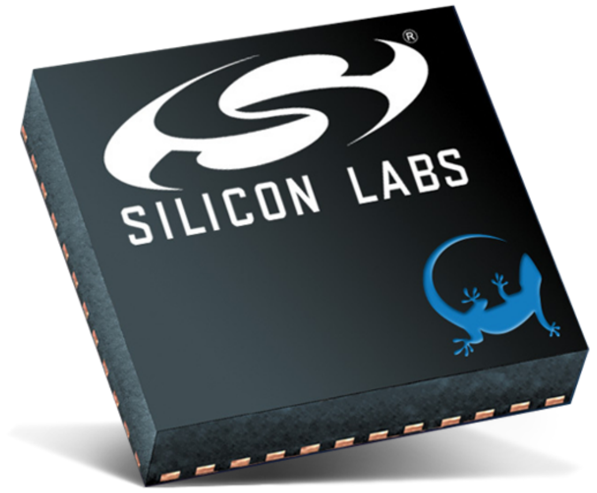


These chips (EFM32/EFR32) are used in a variety of devices including:

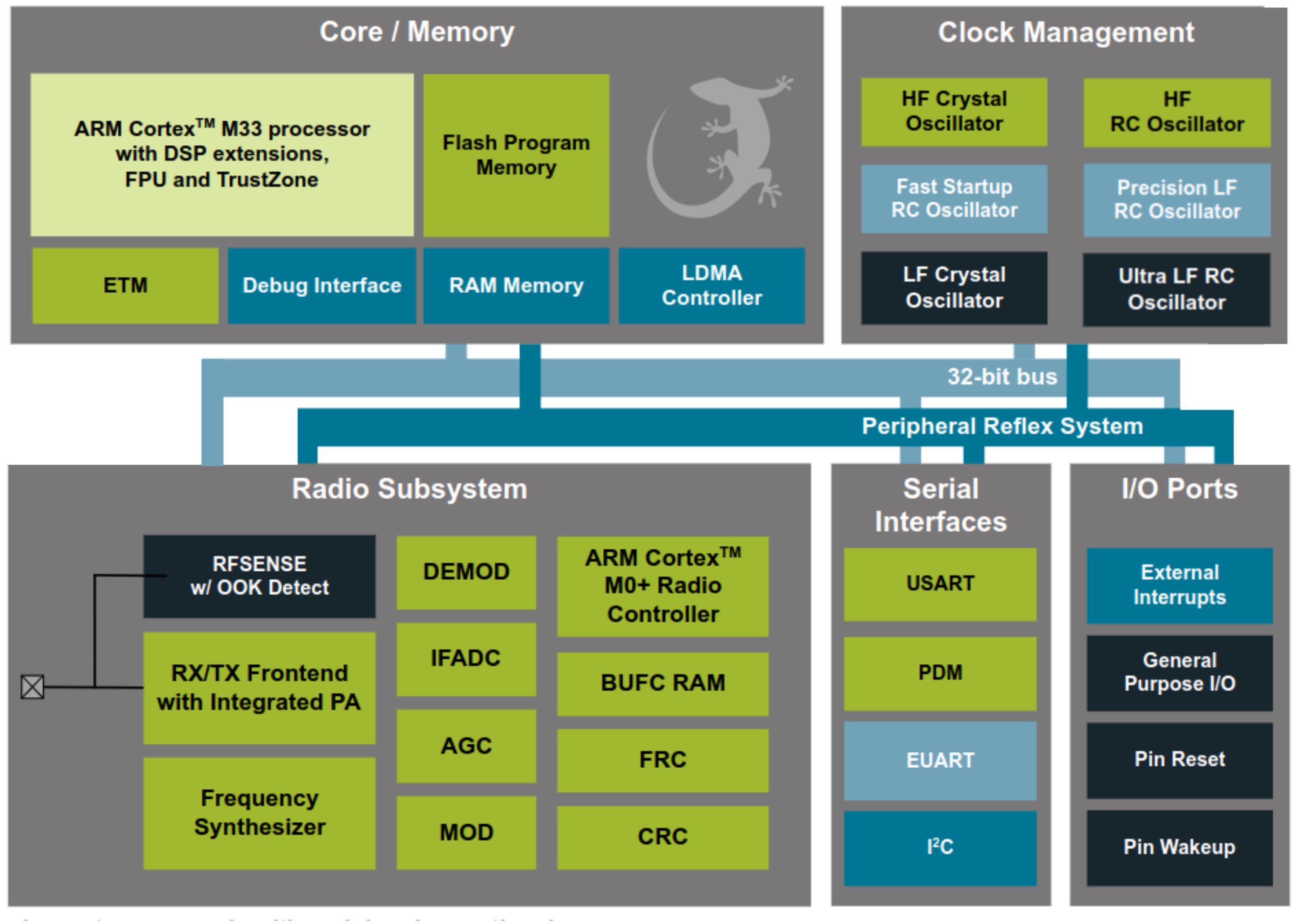
- Amazon Sidewalk.
- Freebox delta.
- Freebox security module.
- …

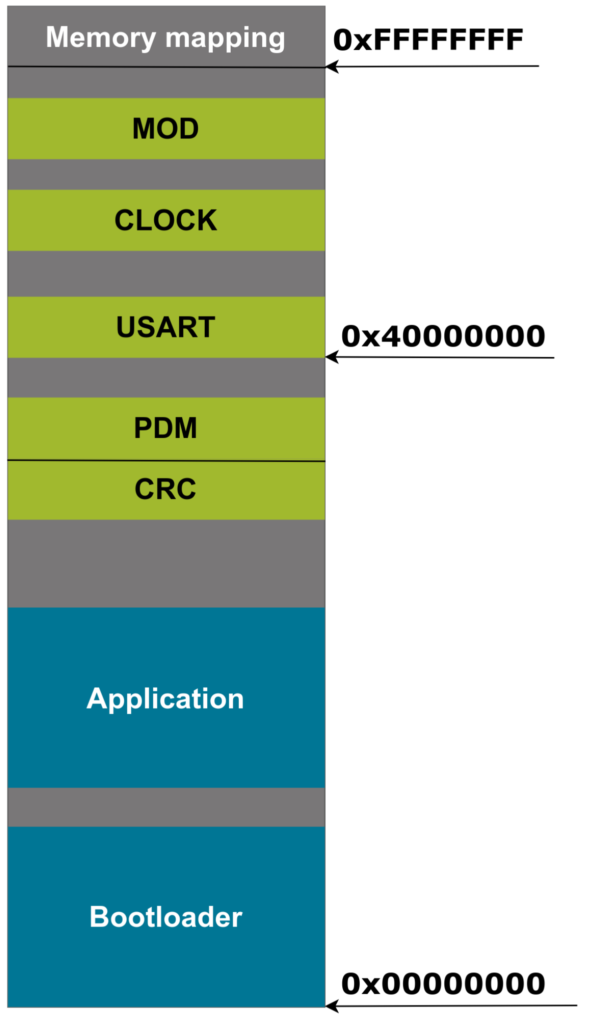
The objective of the research was to look into the SDK offered by Silicon Labs Gecko SDK (GSDK) on:

- Update mechanism (OTA).
- Radio library (RAIL).



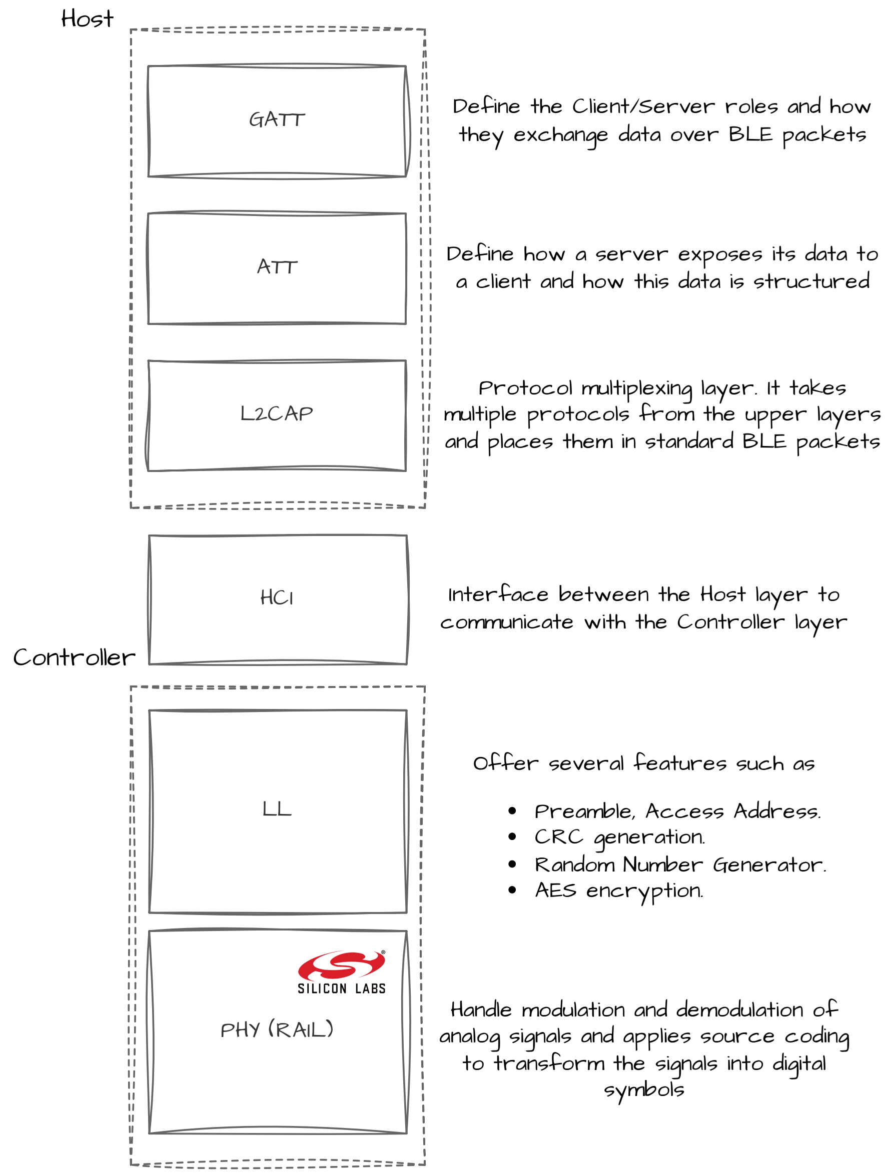
# EFR32 Overview





# Bluetooth Low Energy

- One of the most used wireless protocol in embedded devices.
- Slower than "Classic" Bluetooth but less power consumption.
- 255 bytes at most per packet.
- Client/Server model.



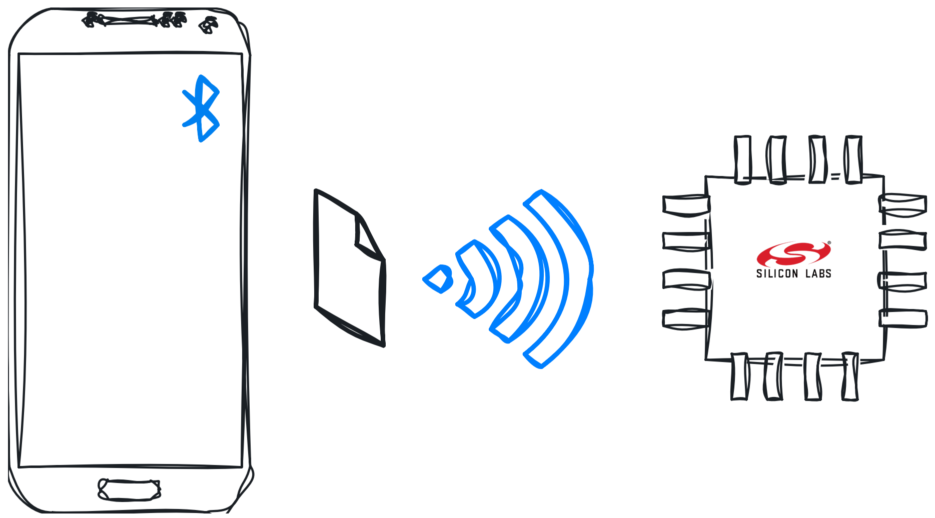
Host
GATT
Define the Client/Server roles and how they exchange data over BLE packets
ATT
Define how a server exposes its data to a client and how this data is structured
L2CAP
Protocol multiplexing layer. It takes multiple protocols from the upper layers and places them in standard BLE packets
HCI
Interface between the Host layer to communicate with the Controller layer
Controller
LL
Offer several features such as
Preamble, Access Address.
CRC generation.
Random Number Generator.
AES encryption.
SILICON LABS
PHY (RAIL)
Handle modulation and demodulation of analog signals and applies source coding to transform the signals into digital symbols

## The OTA update

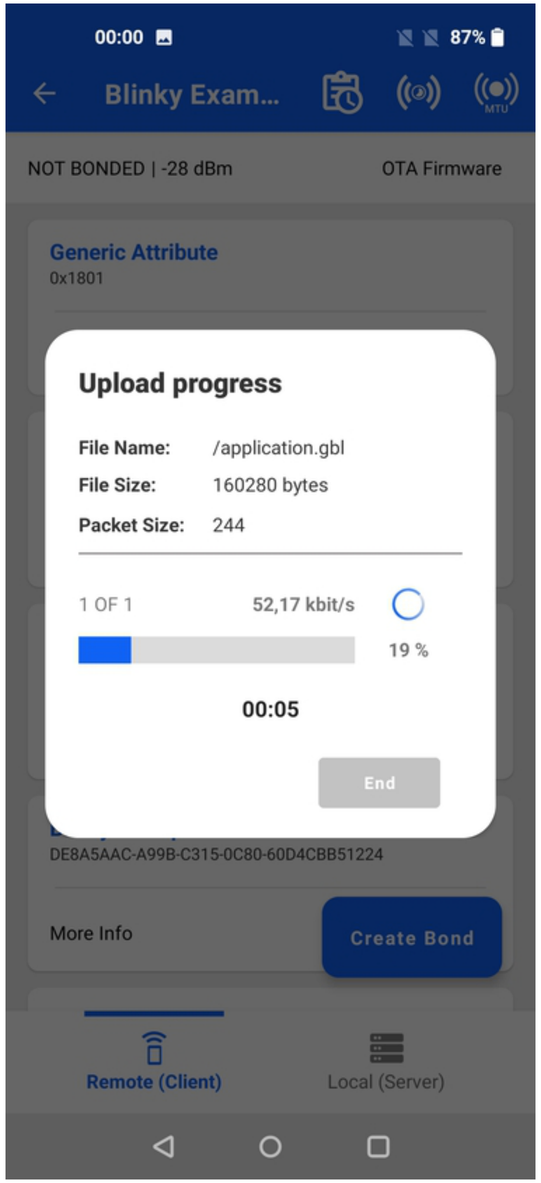
An Over-The-Air (OTA) is an update to an embedded system through a communication protocol (Serial, BLE, Ethernet).

- Allows updates to be distributed at large scales.
- Reduces the cost of delivering updates.
- Increases the rate of adoption of these updates.

Silicon Labs created a custom file format for their firmware called GBL.



# How to upload a GBL file

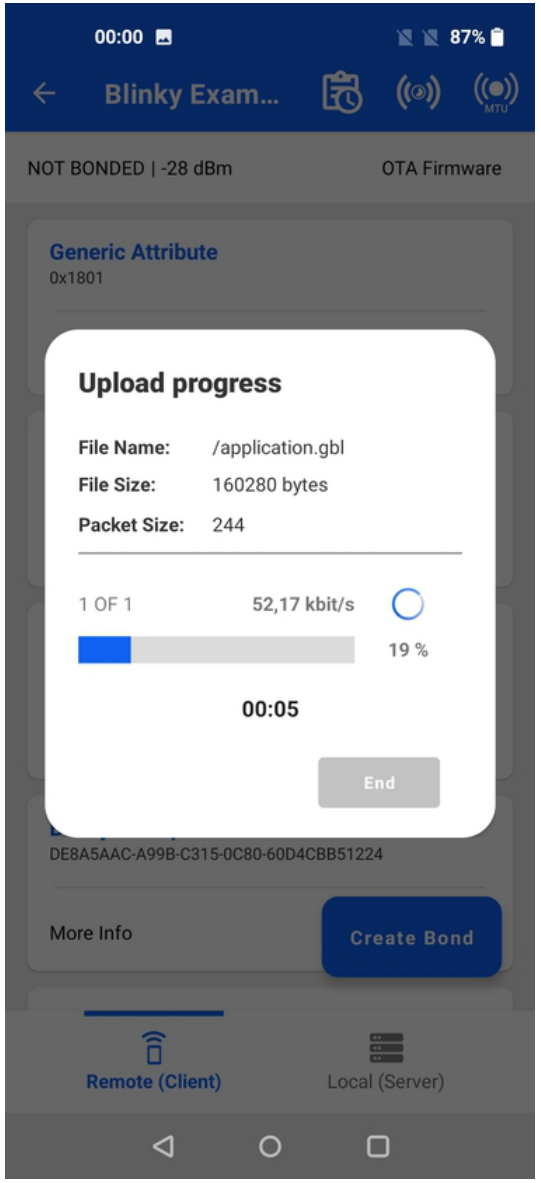


**Upload progress**

**File Name:** /application.gbl
**File Size:** 160280 bytes
**Packet Size:** 244

1 OF 1 52,17 kbit/s
19 %
00:05

# How to upload a GBL file



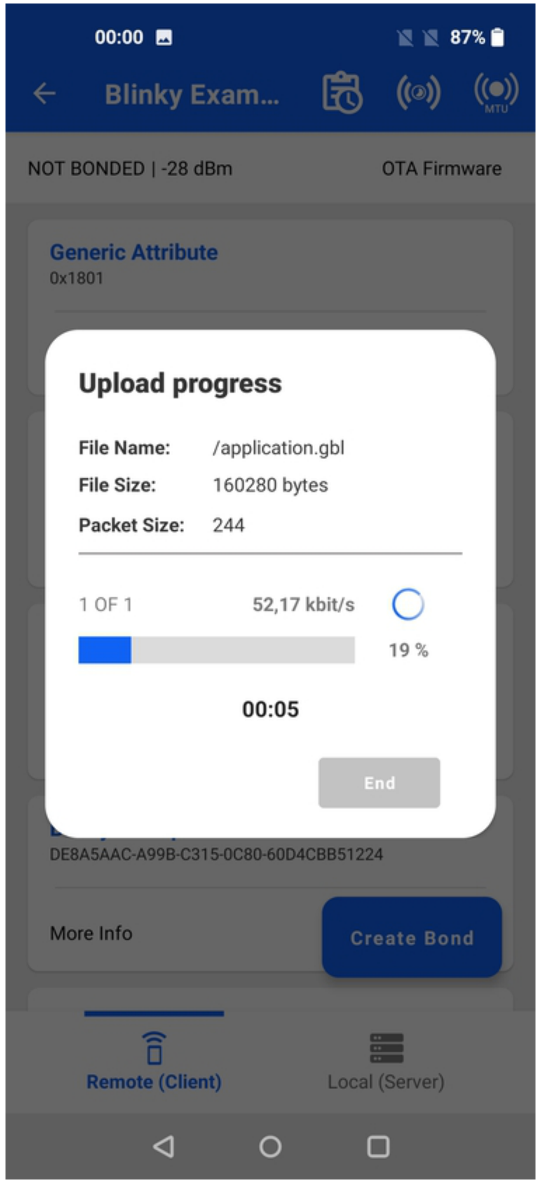
**Upload progress**

File Name: /application.gbl
File Size: 160280 bytes
Packet Size: 244

1 OF 1 52,17 kbit/s
19 %
00:05

# How to upload a GBL file

Install EFR Connect
Use the application with GBL file
The application doesn't work



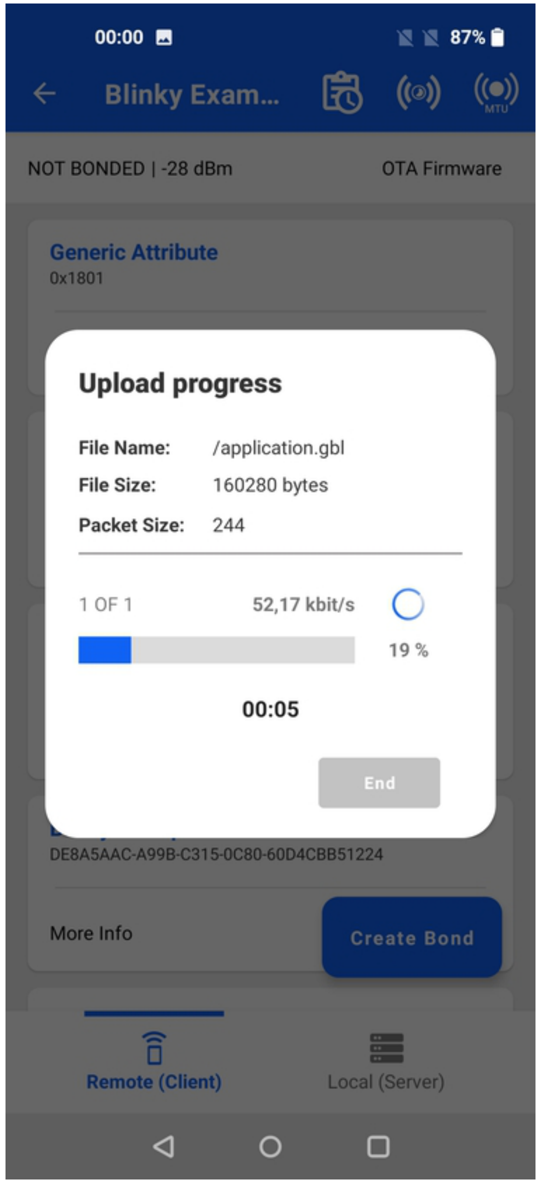
Upload progress

File Name: /application.gbl
File Size: 160280 bytes
Packet Size: 244

1 OF 1 52,17 kbit/s
19 %
00:05

# How to upload a GBL file

Install EFR Connect
Use the application with GBL file
The application doesn't work
The application doesn't work


00:00 87%

Blinky Exam...

NOT BONDED | -28 dBm OTA Firmware

Generic Attribute
0x1801

Upload progress

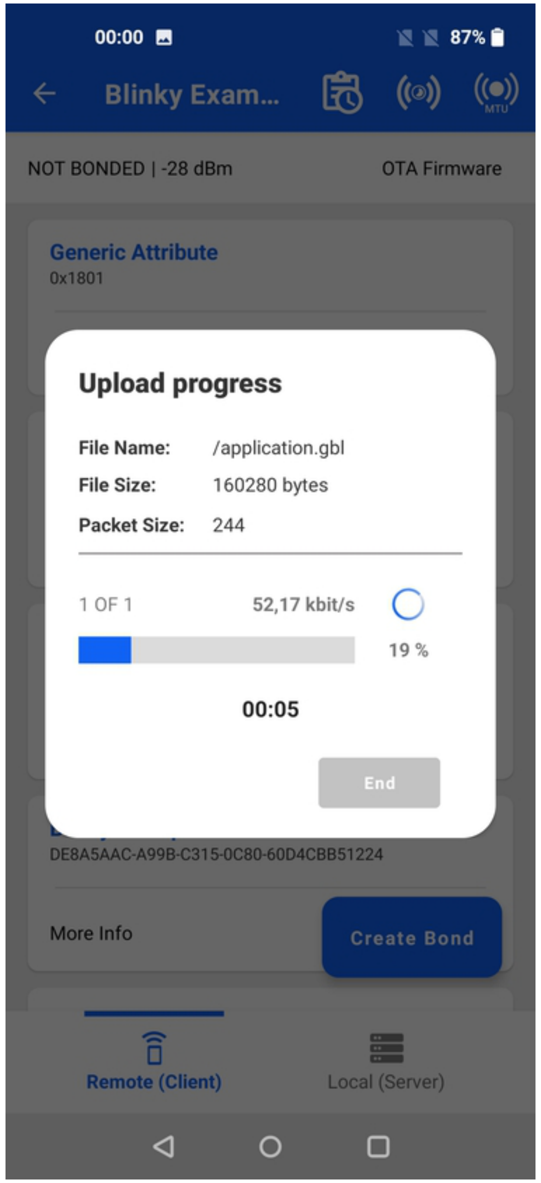
File Name: /application.gbl
File Size: 160280 bytes
Packet Size: 244

1 OF 1 52,17 kbit/s
19 %
00:05

DE8A5AAC-A99B-C315-0C80-60D4CBB51224

# How to upload a GBL file

Install EFR Connect
Use the application with GBL file
The application doesn't work
Reverse the application



NOT BONDED | -28 dBm    OTA Firmware

Generic Attribute
0x1801

Upload progress

File Name: /application.gbl
File Size: 160280 bytes
Packet Size: 244

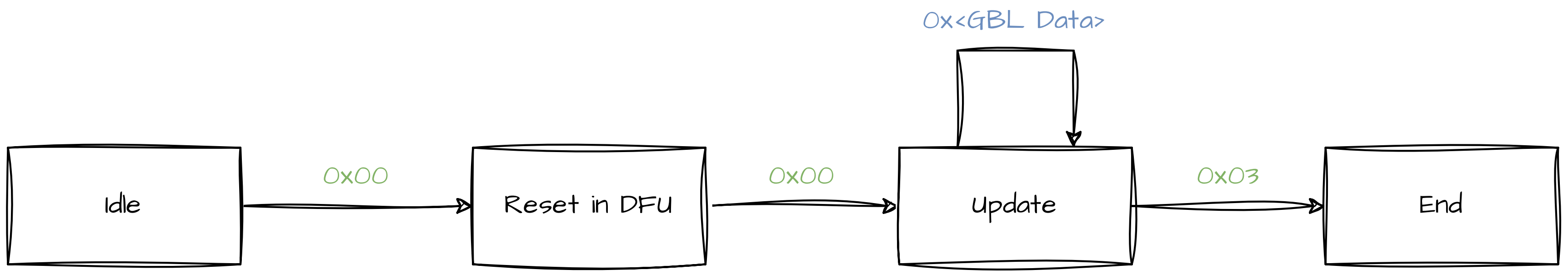
1 OF 1    52,17 kbit/s
19 %
00:05

DE8A5AAC-A99B-C315-0C80-60D4CBB51224

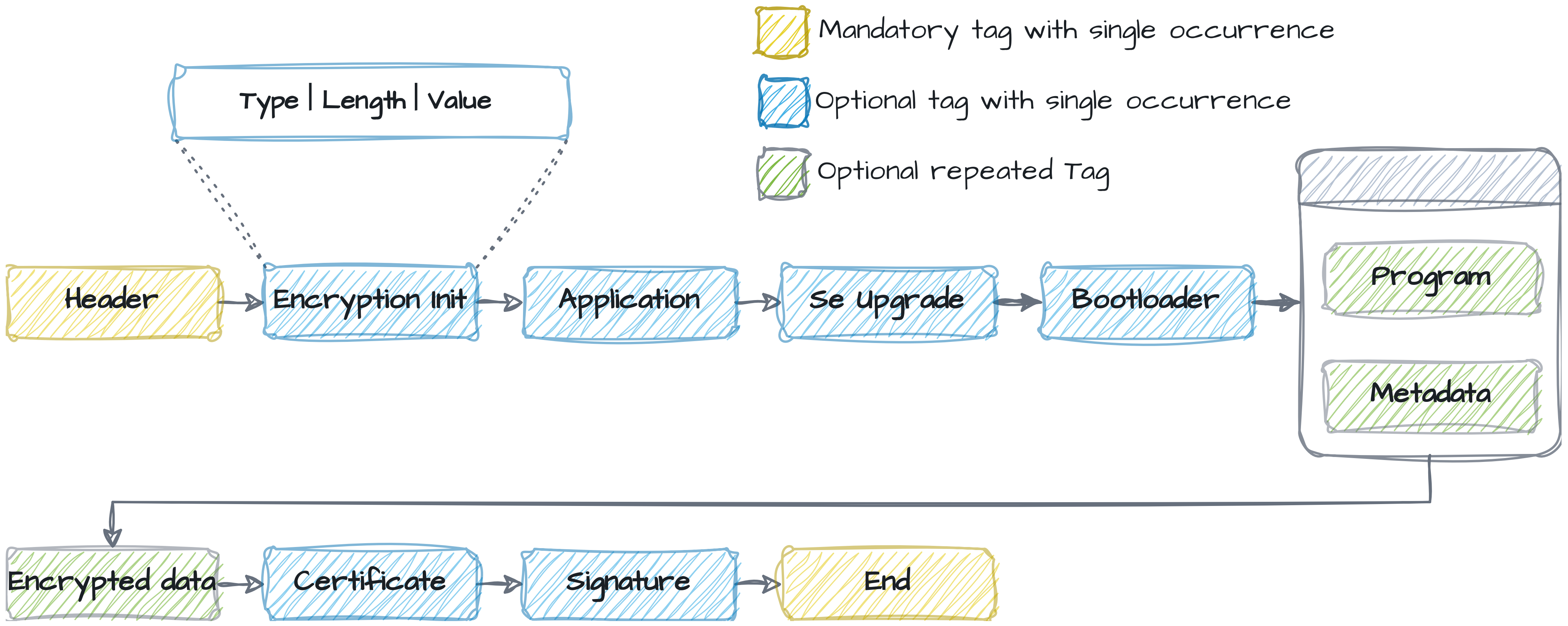
# How to upload a GBL file

OTA_DATA_UUID 984227f3-34fc-4045-a5d0-2c581f81a153

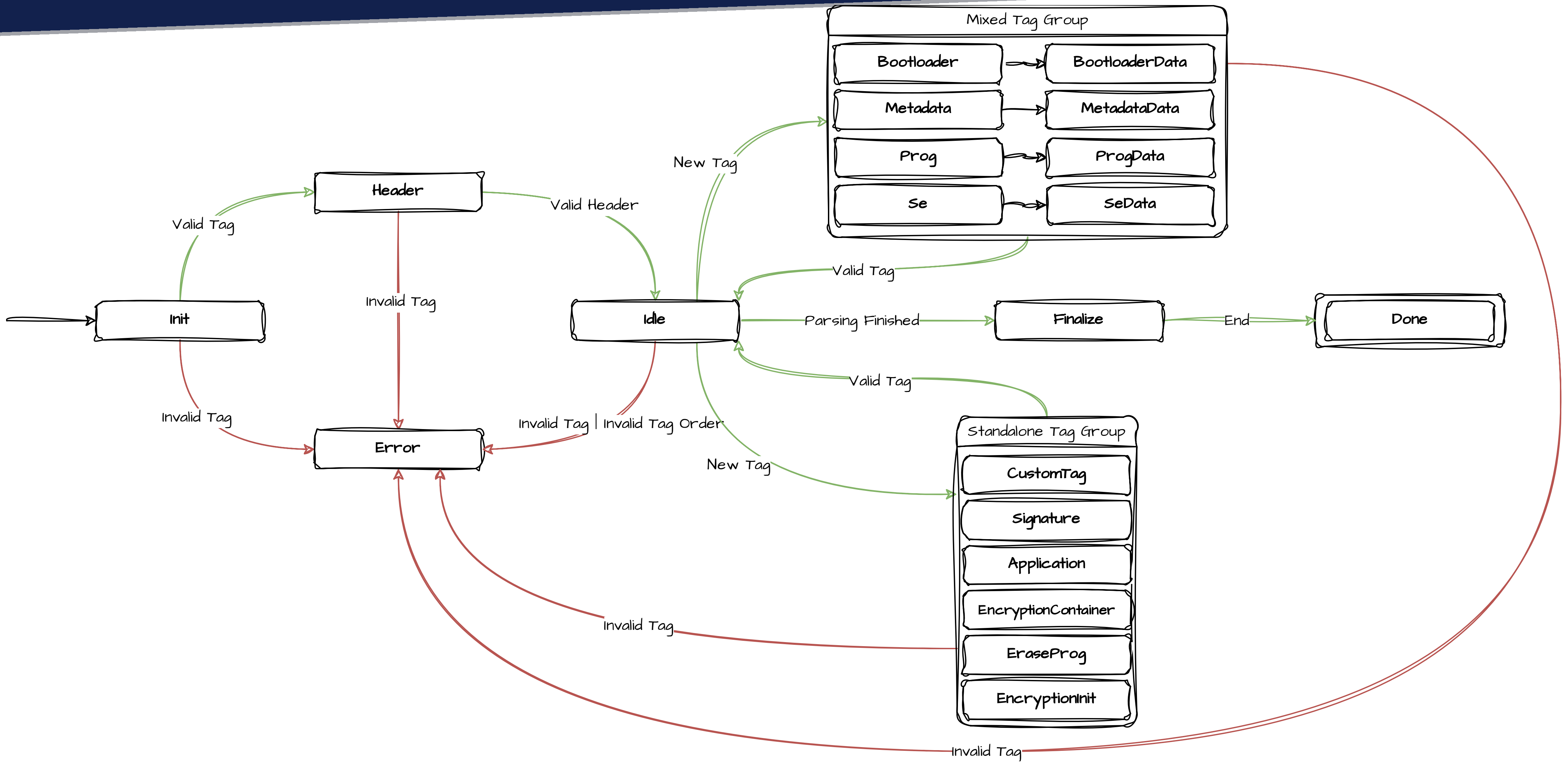
OTA_CONTROL_UUID f7bf3564-fb6d-4e53-88a4-5e37e0326063



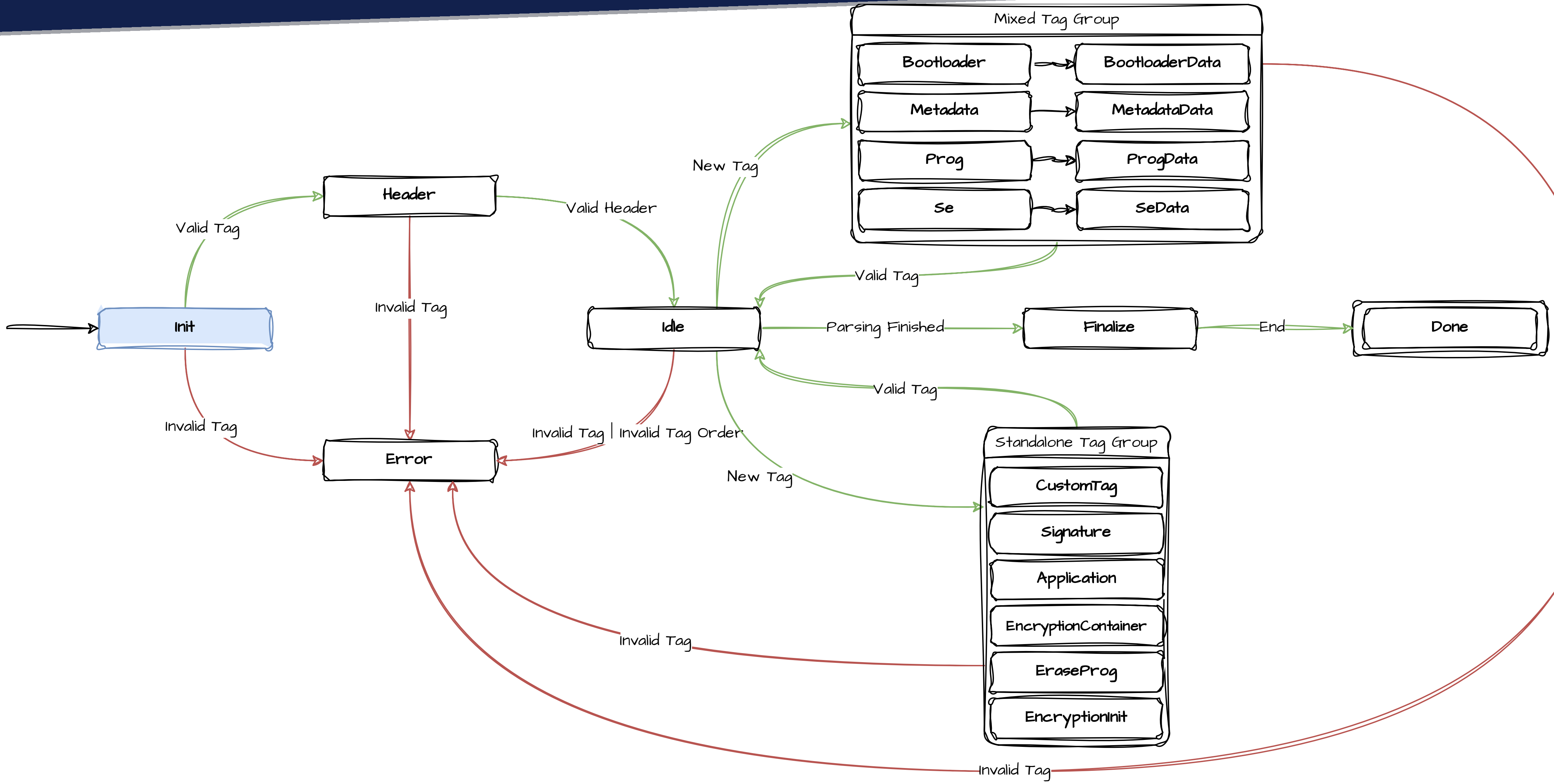
# The GBL file format



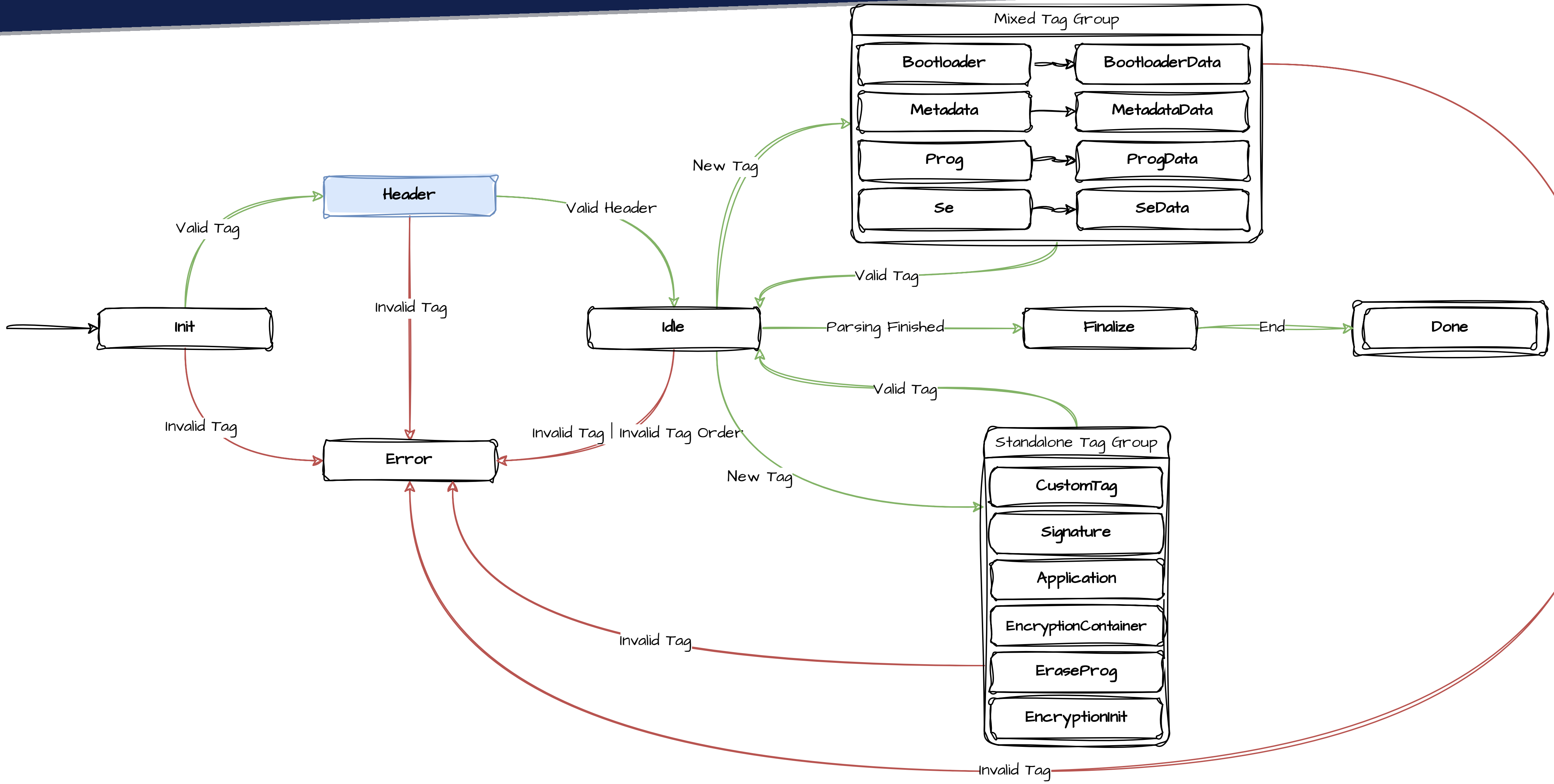
# The OTA State Machine



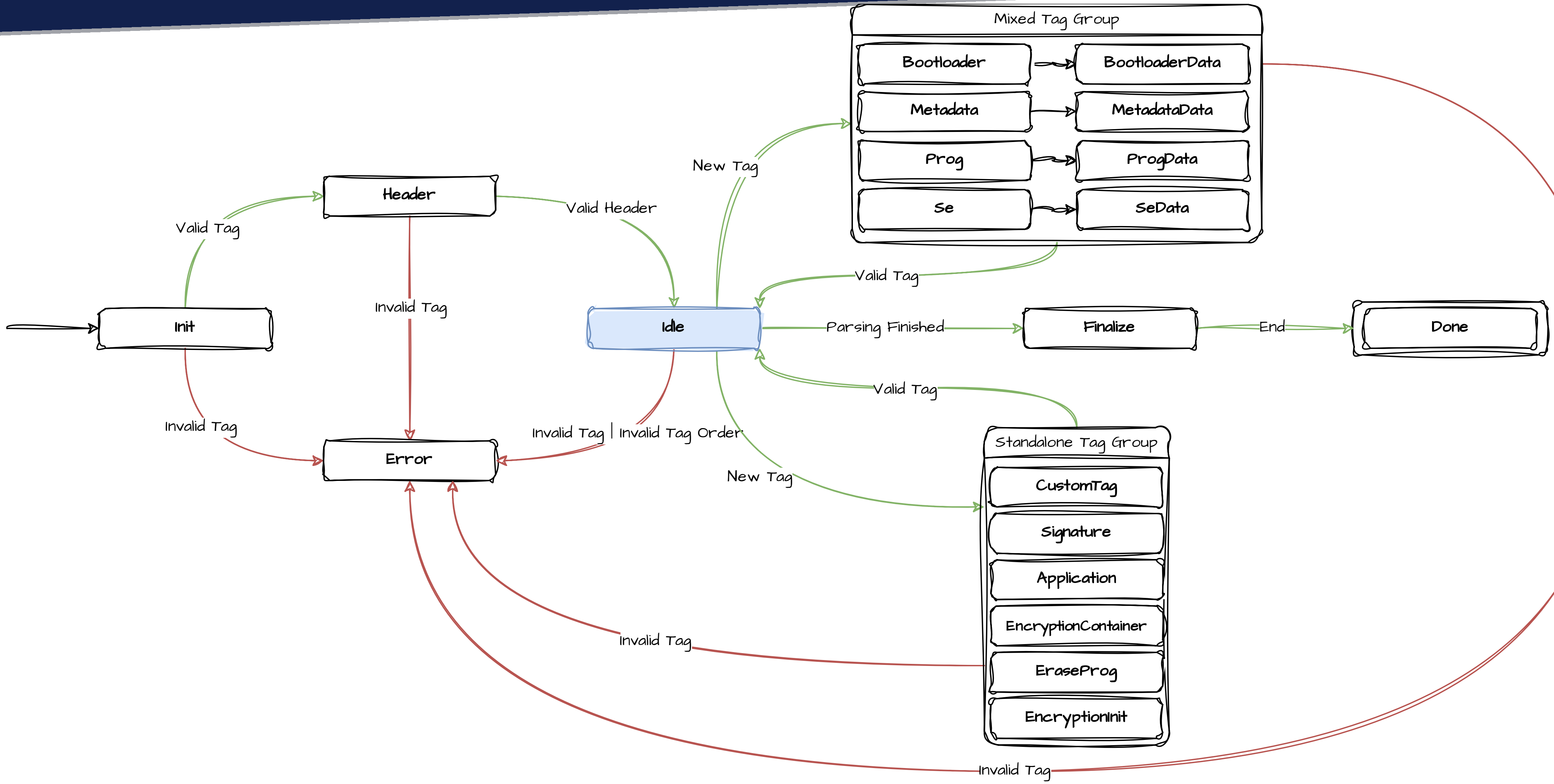
# The OTA State Machine



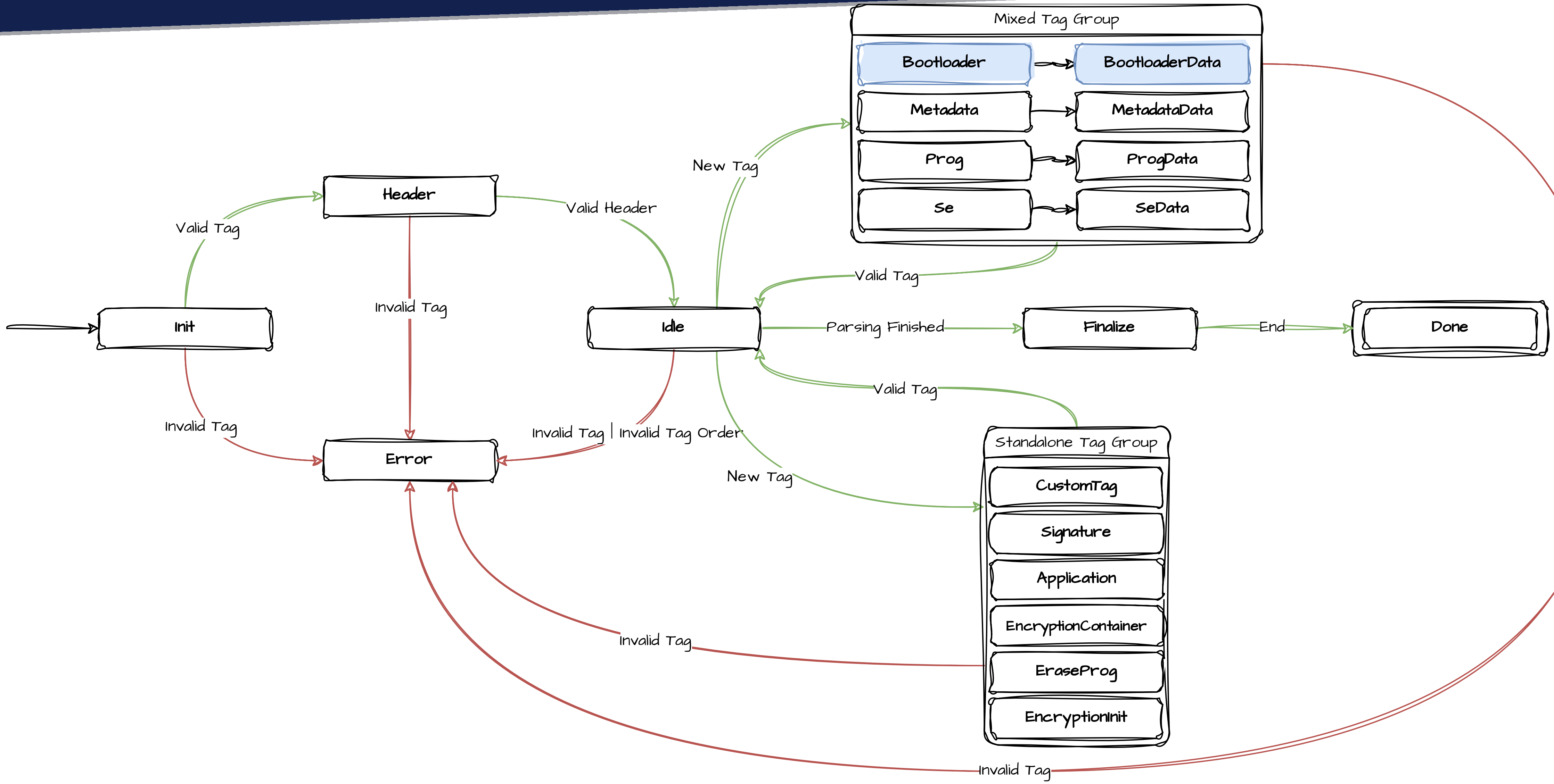
# The OTA State Machine



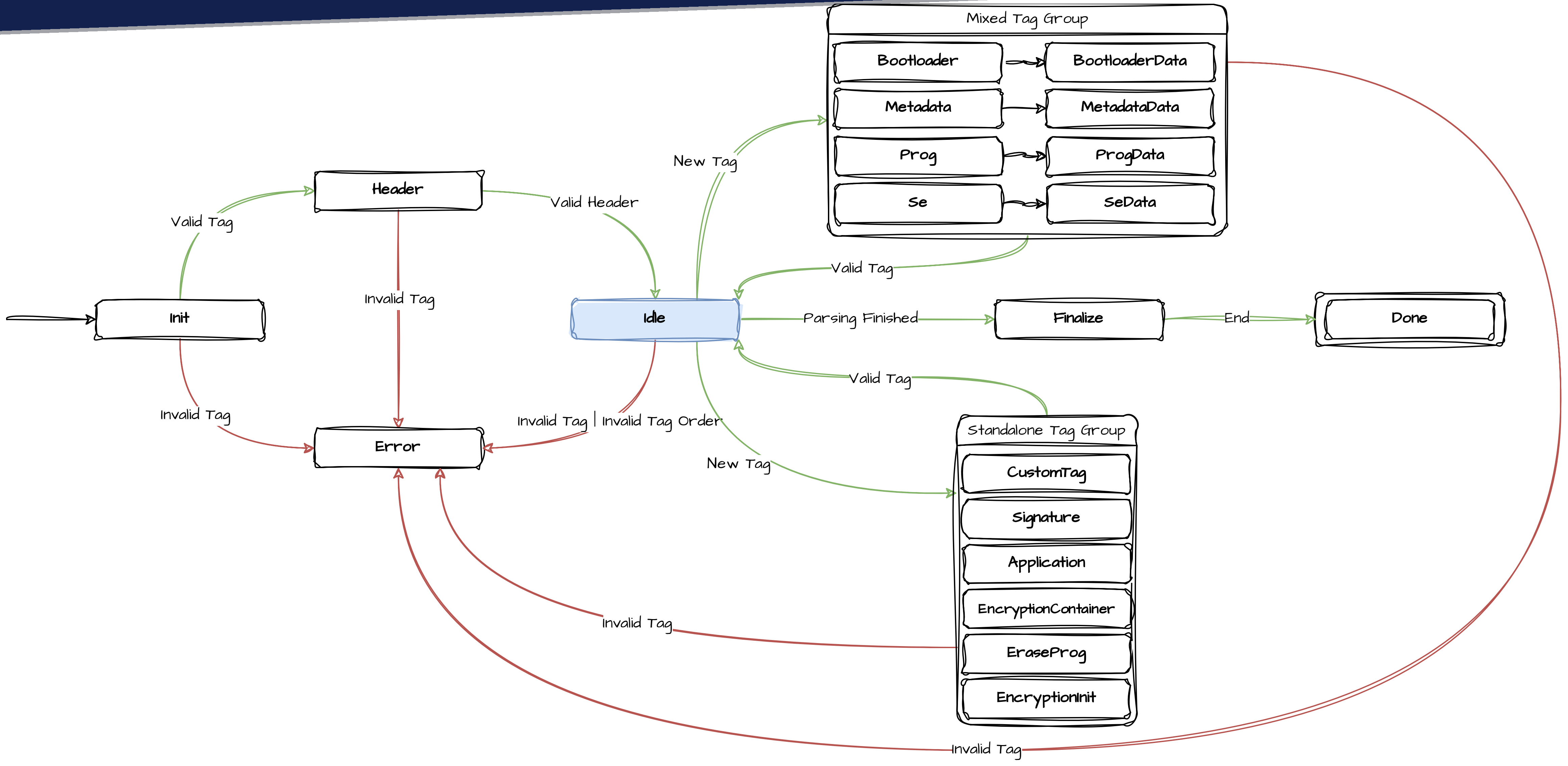
# The OTA State Machine



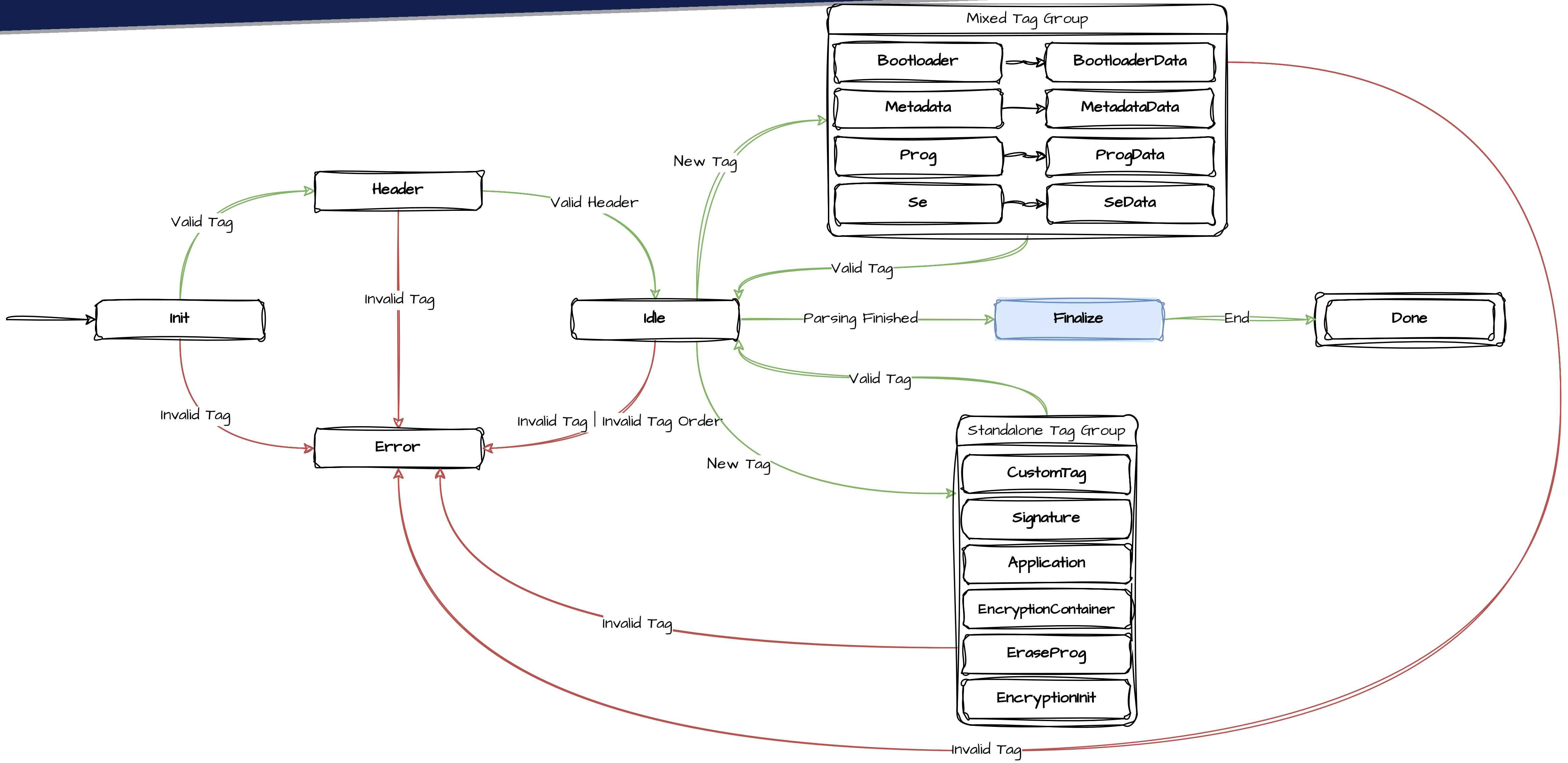
# The OTA State Machine



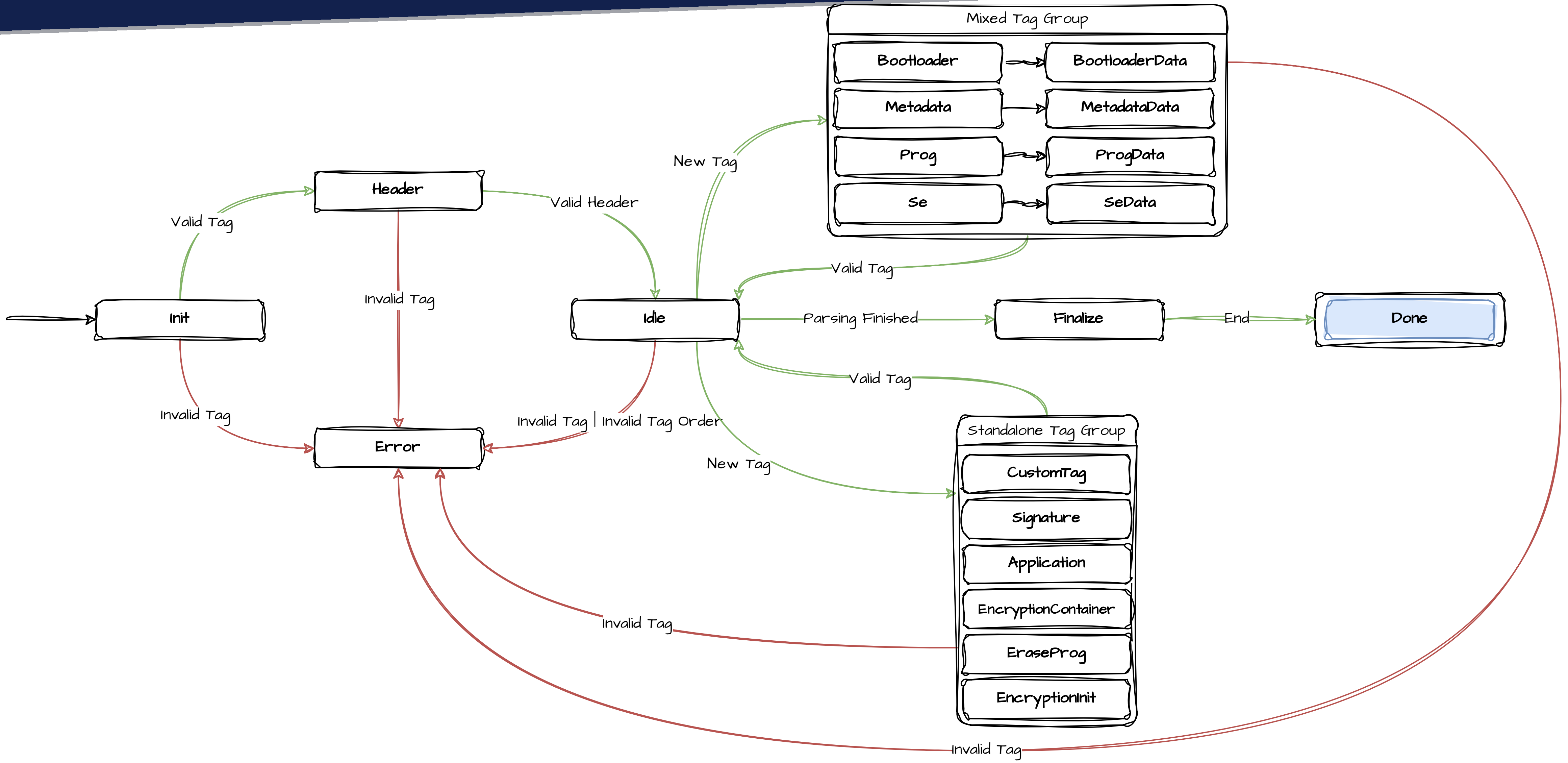
# The OTA State Machine



# The OTA State Machine



# The OTA State Machine



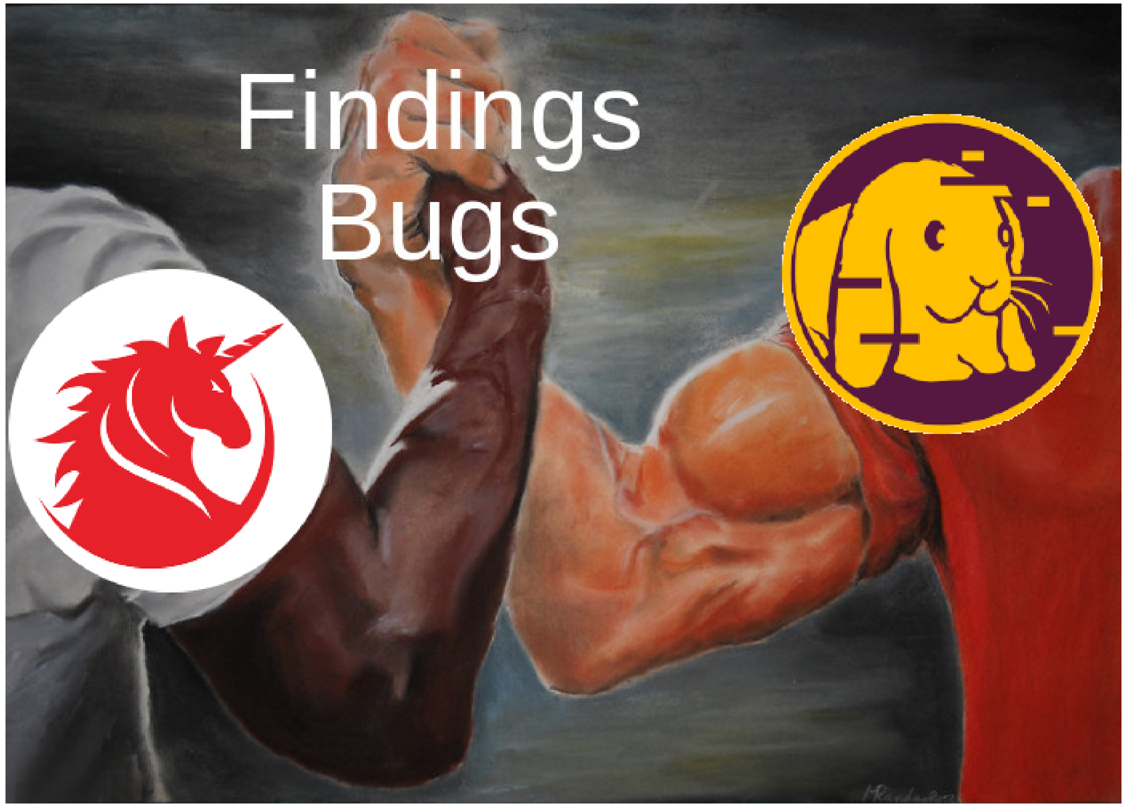
## Vulnerability research: Fuzzing Approach

Why choose a fuzzing-oriented approach?

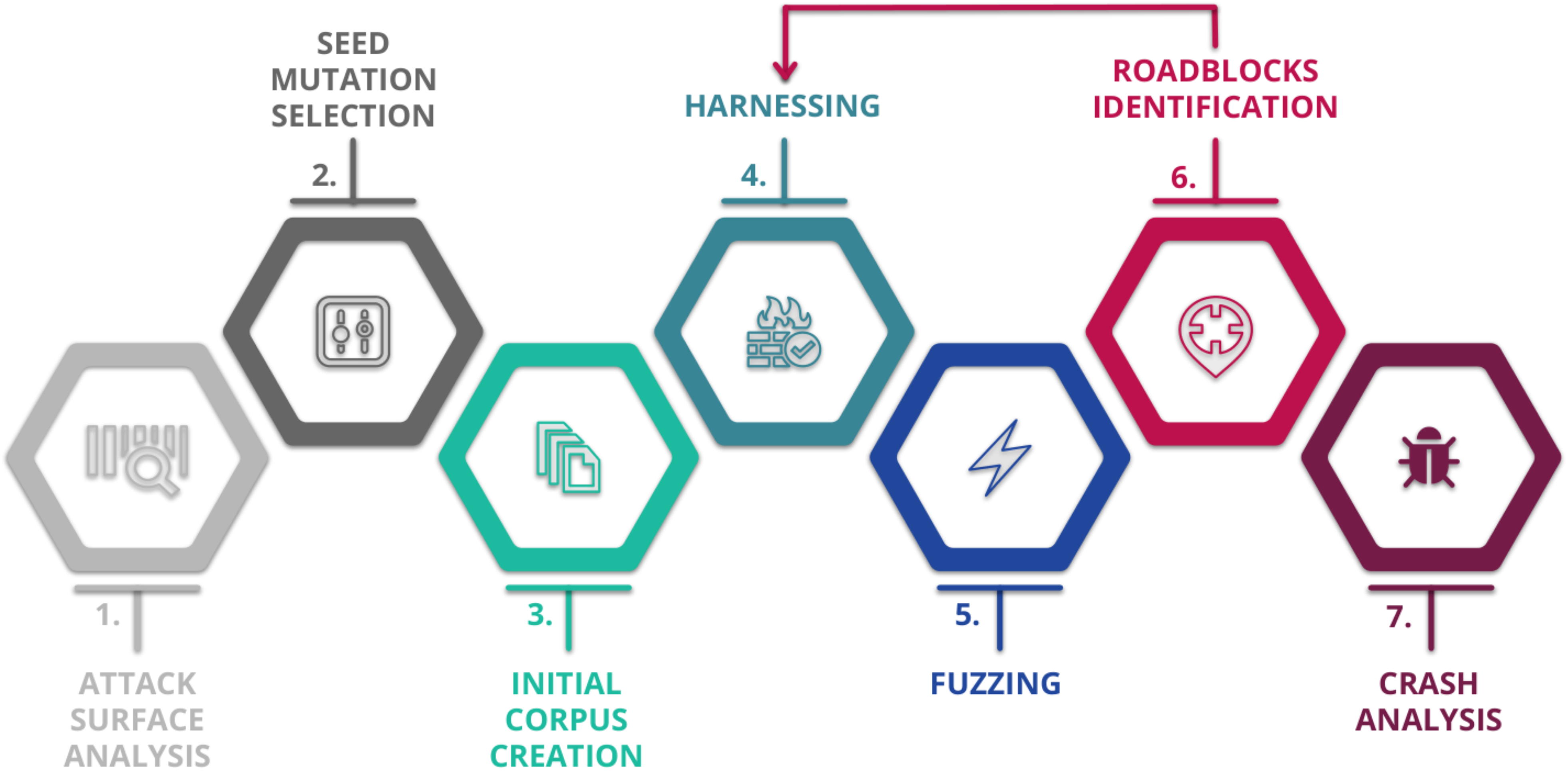
- It is automatic and efficient.
- A lot of fuzzing experts at Quarkslab.
- For learning purposes.

What do we need?

- Be able to run ARM code on the computer.
- Generate interesting inputs.
- Last but not least a fuzzer.



# Fuzzing overview



## Protocol Buffers & Custom Mutators

```
def mutate_gbl(gbl):
    # List of all the mutations we can do
    mutations = [
        mutate_add_tag, mutate_remove_tag,
        mutate_tag, mutate_reset_tag,
        mutate_reset_gbl
    ]
    weights = [
        0.15, 0.15, 0.55, 0.01, 0.05
    ]
    # Choice a random mutation
    mutation = numpy.random.choice(mutations, p=weights)
    # Apply mutation
    return mutation(gbl)
```

```
message ApplicationData_t {
    // Bitfield representing type of application
    optional uint32 type = 1;
    // Version number for this application
    optional uint32 version = 2;
    // Capabilities of this application
    optional uint32 capabilities = 3;
    // Unique ID (UUID or GUID) for the product this application is built for
    // uint8 types in protobuf.
    optional fixed64 productId_upper = 4;
    optional fixed64 productId_lower = 5;
}
```

Protobuf is a mechanism for serializing structured data.

- Fast & Simple.
- Integrated with AFL++.

AFL++ custom mutators offer better control over the mutations.

## Emulating code

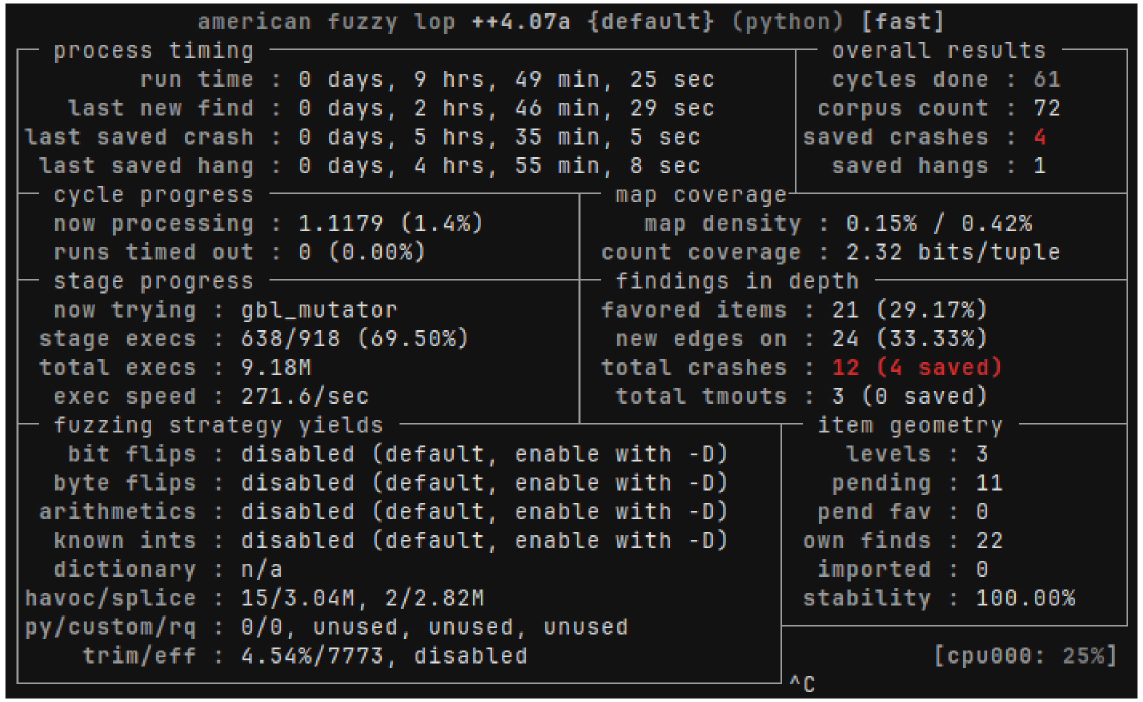
Use Unicorn (framework based on QEMU) to emulate the code of the parser:

- Need to correctly setup all the memory:
  - Bootloader and application zone;
  - Peripherals;
  - Stack.
- A lot of patches (custom hooks):
  - Cryptographic functions hardware based;
  - Problematics check.

```
# void * __stdcall memcpy(void * __dest, void * __src, size_t __n)
# void *            r0:4           <RETURN>
# void *            r0:4           __dest
# void *            r1:4           __src
# size_t            r2:4           __n
def memcpy(emu, address, size, user_data):
    # Retrieve our argument(s)
    dest = emu.reg_read(emu.ARM_REG_R0)
    src  = emu.reg_read(emu.ARM_REG_R1)
    size = emu.reg_read(emu.ARM_REG_R2)
    # memcpy
    emu.mem_write(dest, bytes(emu.mem_read(src, size)))
    # return to caller (skip the function)
    emu.reg_write(emu.ARM_REG_PC, emu.reg_read(emu.ARM_REG_LR))
```

```
# void __stdcall btl_updateSha256(void *ctx,void *data,size_t length)
# void              <VOID>         <RETURN>
# void *            r0:4           ctx
# void *            r1:4           data
# size_t            r2:4           length
def btl_updateSha256(emu, address, size, user_data):
    sha256 = user_data['handle']
    # Retrieve our argument(s)
    data   = emu.reg_read(emu.ARM_REG_R1)
    length = emu.reg_read(emu.ARM_REG_R2)
    sha256.update(emu.mem_read(data, length))
    emu.reg_write(emu.ARM_REG_PC, emu.reg_read(emu.ARM_REG_LR))
```

# Fuzzing the OTA parser



```
          american fuzzy lop ++4.07a {default} (python) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 9 hrs, 49 min, 25 sec     │  cycles done : 61    │
│   last new find : 0 days, 2 hrs, 46 min, 29 sec     │ corpus count : 72    │
│last saved crash : 0 days, 5 hrs, 35 min, 5 sec      │saved crashes : 4     │
│ last saved hang : 0 days, 4 hrs, 55 min, 8 sec      │  saved hangs : 1     │
├─ cycle progress ───────────────────┬─ map coverage─┴──────────────────────┤
│  now processing : 1.1179 (1.4%)    │    map density : 0.15% / 0.42%       │
│  runs timed out : 0 (0.00%)        │ count coverage : 2.32 bits/tuple     │
├─ stage progress ───────────────────┼─ findings in depth ──────────────────┤
│  now trying : gbl_mutator          │ favored items : 21 (29.17%)          │
│ stage execs : 638/918 (69.50%)     │  new edges on : 24 (33.33%)          │
│ total execs : 9.18M                │ total crashes : 12 (4 saved)         │
│  exec speed : 271.6/sec            │  total tmouts : 3 (0 saved)          │
├─ fuzzing strategy yields ──────────┴─────────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D) │    levels : 3         │
│  byte flips : disabled (default, enable with -D) │   pending : 11        │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 0         │
│  known ints : disabled (default, enable with -D) │ own finds : 22        │
│  dictionary : n/a                                │  imported : 0         │
│havoc/splice : 15/3.04M, 2/2.82M                  │ stability : 100.00%   │
│py/custom/rq : 0/0, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 4.54%/7773, disabled               │          [cpu000: 25%]
└──────────────────────────────────────────────────┘^C
```

After multiple fuzzing campaigns for a total of 10h AFL++ found 4 crashes:

- Two of them are false positives (crash of the emulator).
- The third one is not reproducible on the target.
- The last one is promising.

# CVE-2023-4041

```
static int32_t parser_parseApplicationInfo(ParserContext_t   *parserContext,
                                           GblInputBuffer_t  *input,
                                           ImageProperties_t *imageProperties)
{
  volatile int32_t retval;
  uint8_t tagBuffer[GBL_PARSER_BUFFER_SIZE];

  while (parserContext->offsetInTag < parserContext->lengthOfTag) {
    // Get data
    retval = gbl_getData(parserContext,
                         input,
                         tagBuffer,
                         parserContext->lengthOfTag,
                         true,
                         true);
```

There is a buffer overflow inside the parser_parseApplicationInfo function because we control the `parserContext->lengthOfTag`.

## CVE-2023-4041

```
static int32_t parser_parseApplicationInfo(ParserContext_t   *parserContext,
                                           GblInputBuffer_t  *input,
                                           ImageProperties_t *imageProperties)
{
  volatile int32_t retval;
  uint8_t tagBuffer[GBL_PARSER_BUFFER_SIZE];

  while (parserContext->offsetInTag < parserContext->lengthOfTag) {
    // Get data
    retval = gbl_getData(parserContext,
                         input,
                         tagBuffer,
                         parserContext->lengthOfTag,
                         true,
                         true);
```

There is a buffer overflow inside the parser_parseApplicationInfo function because we control the `parserContext->lengthOfTag`.

# CVE-2023-4041

```
static int32_t parser_parseApplicationInfo(ParserContext_t   *parserContext,
                                           GblInputBuffer_t  *input,
                                           ImageProperties_t *imageProperties)
{
  volatile int32_t retval;
  uint8_t tagBuffer[GBL_PARSER_BUFFER_SIZE];

  while (parserContext->offsetInTag < parserContext->lengthOfTag) {
    // Get data
    retval = gbl_getData(parserContext,
                         input,
                         tagBuffer,
                         parserContext->lengthOfTag,
                         true,
                         true);
```

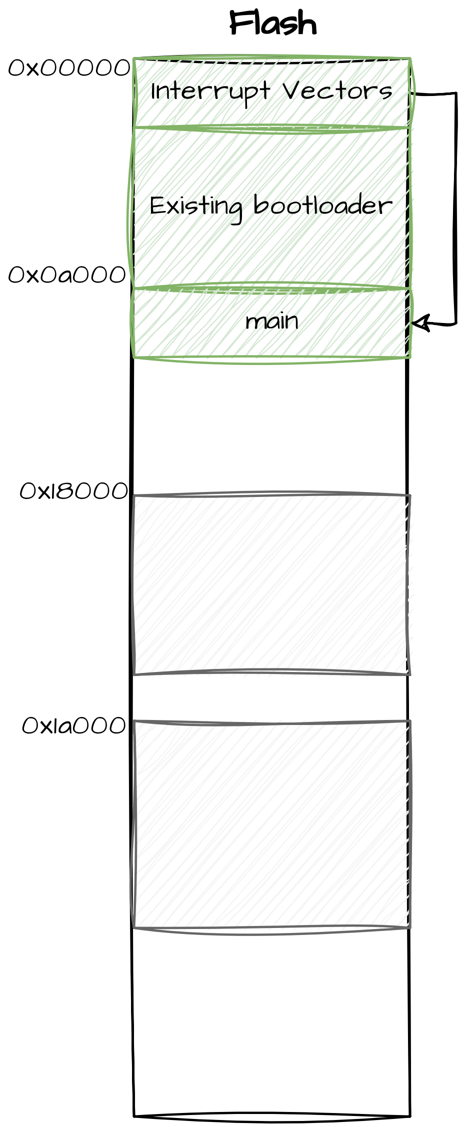
There is a buffer overflow inside the parser_parseApplicationInfo function because we control the `parserContext->lengthOfTag`.

## Demo Time

# Exploit Development

- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

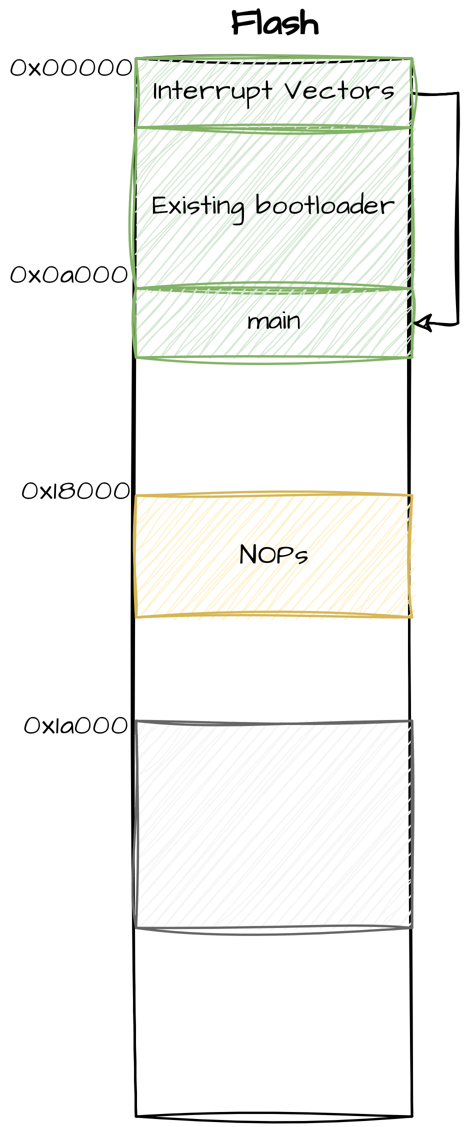
- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

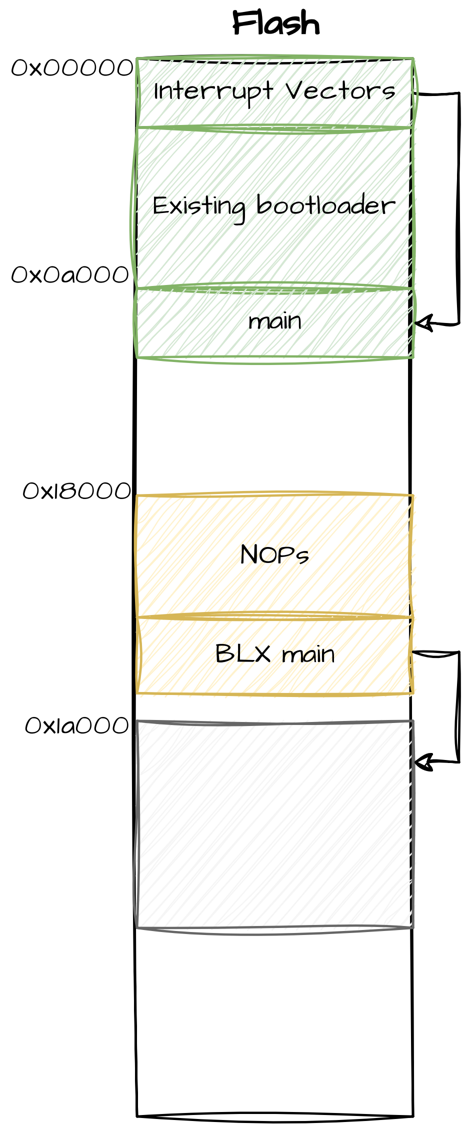
- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

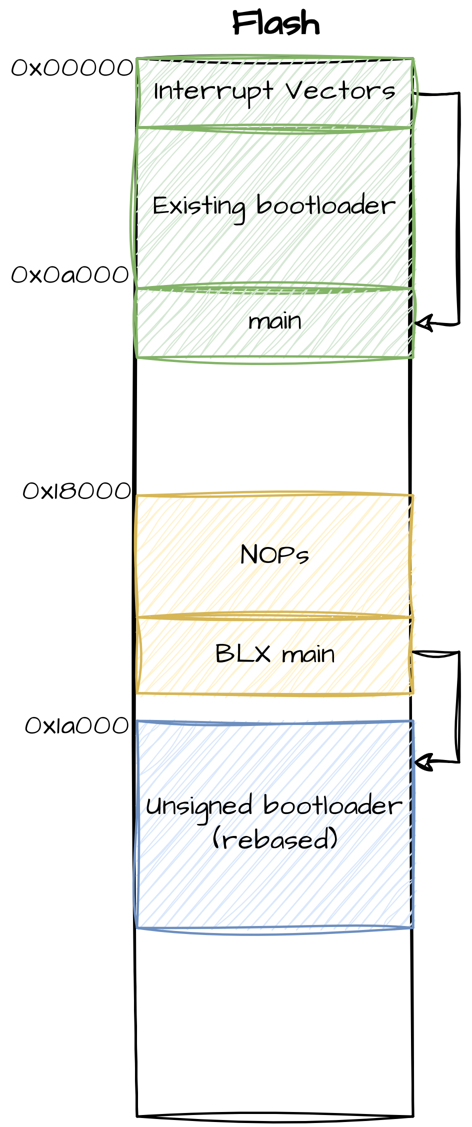
- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ❎ Encryption breaks everything.
- ❎ Limited in size because of BLE packet size.
- ❎ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

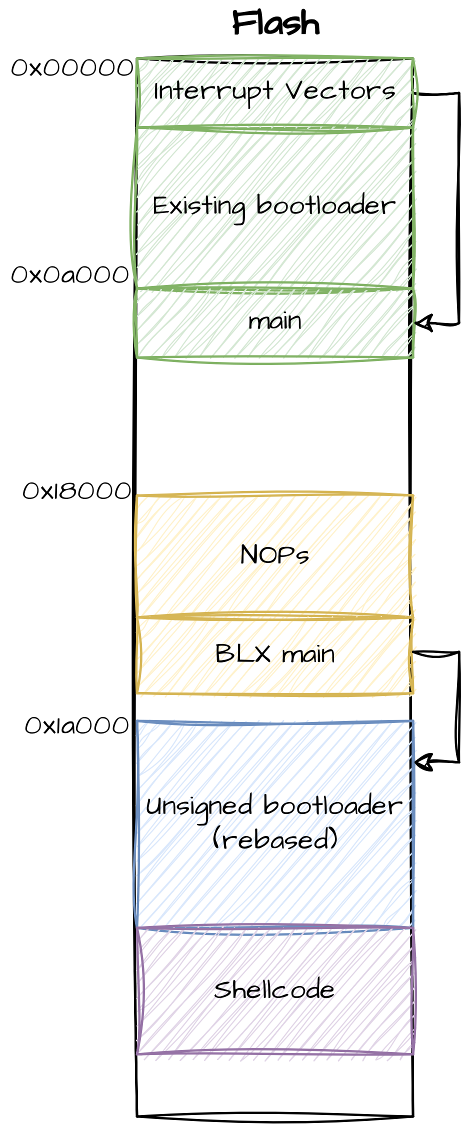
- ✅ Erase and write the page at 0xa000 that contains the main function.
- ✅ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ❌ Encryption breaks everything.
- ❌ Limited in size because of BLE packet size.
- ❌ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

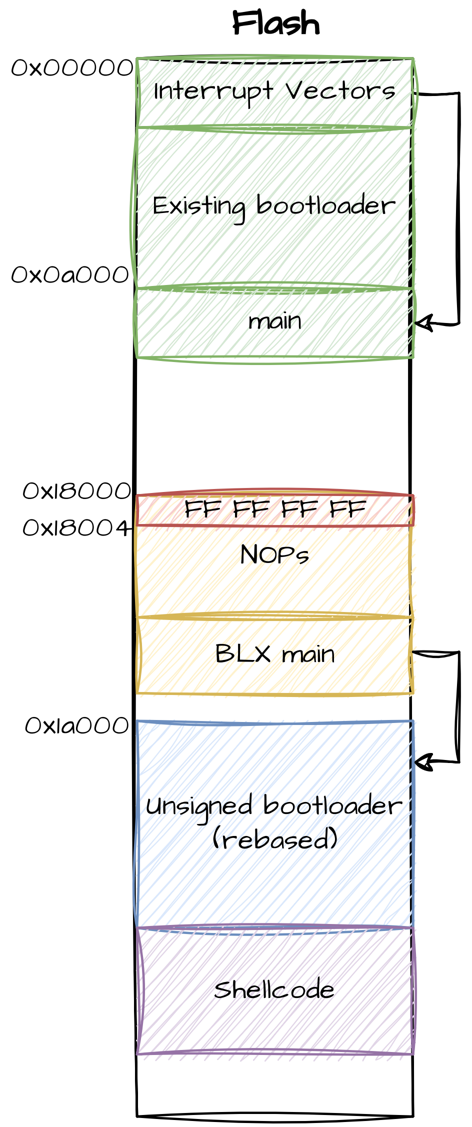
- ✅ Erase and write the page at 0xa000 that contains the main function.
- ✅ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

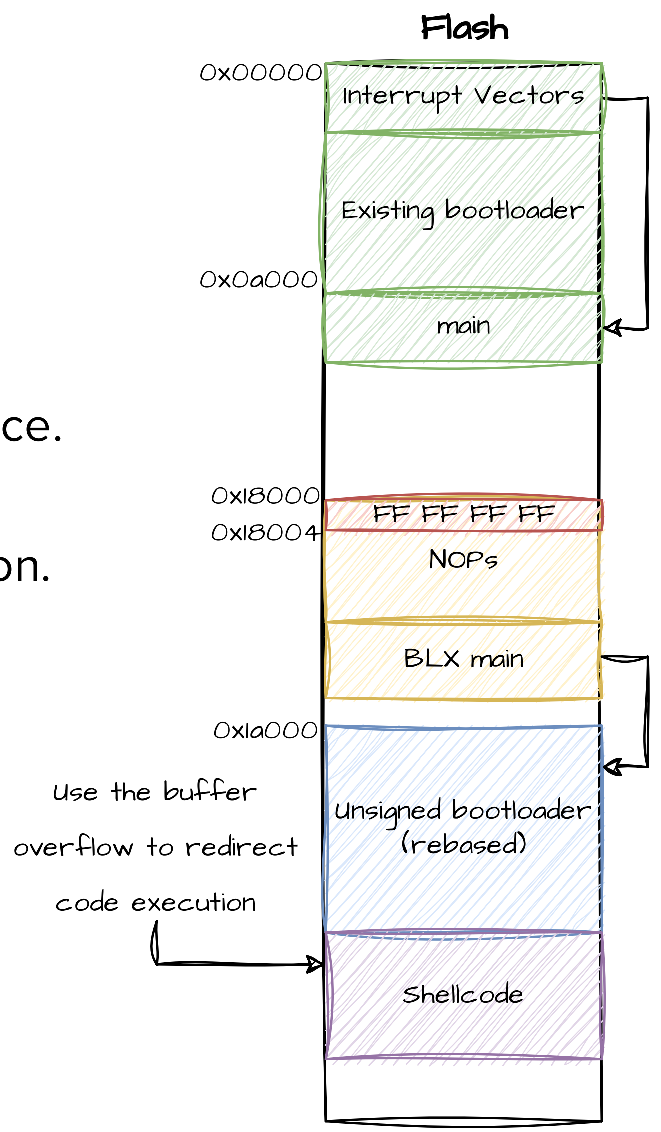
- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



# Exploit Development

- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

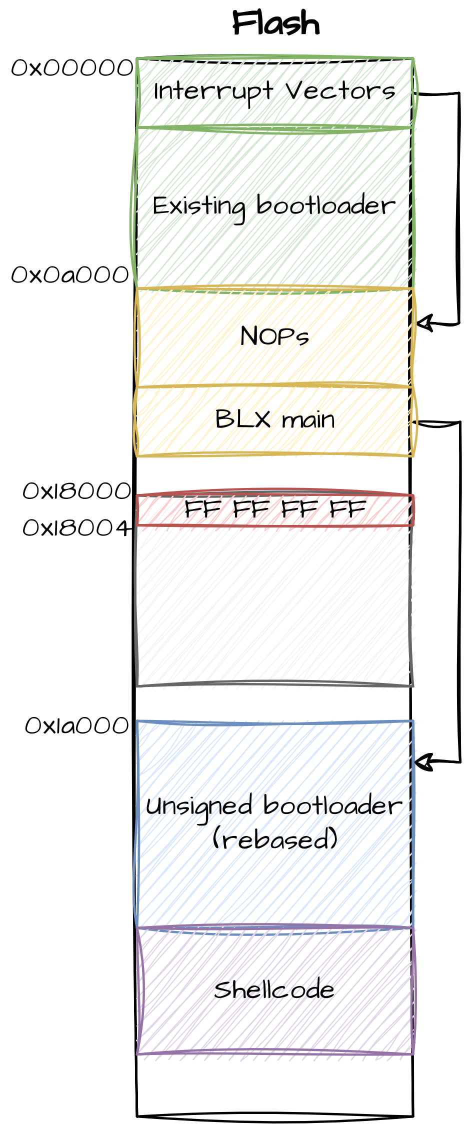
- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



# Exploit Development

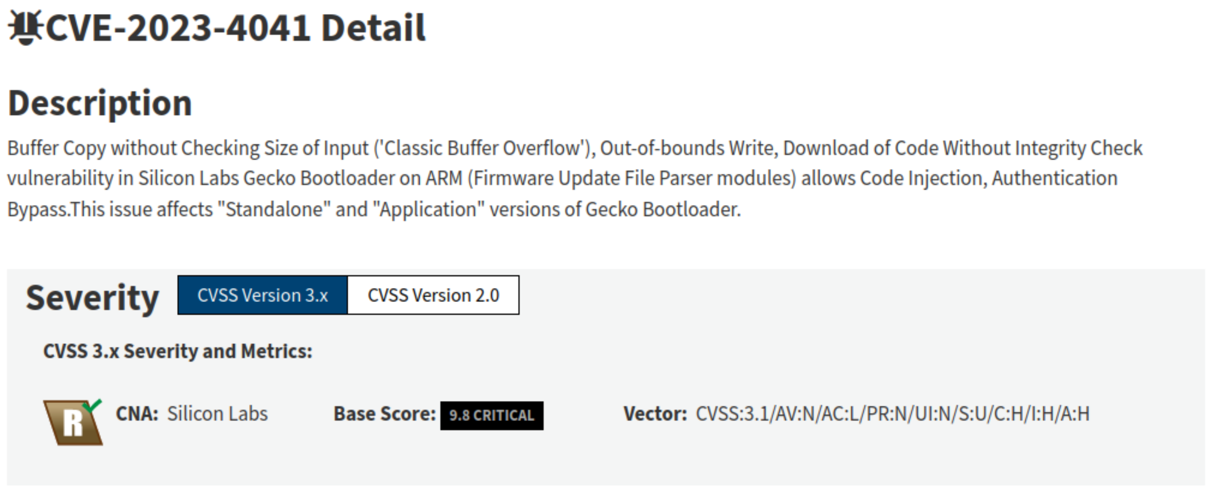
- ✗ Encryption breaks everything.
- ✗ Limited in size because of BLE packet size.
- ✗ Erase pages from 0x0 to 0x8000 otherwise we will brick the device.

- ✓ Erase and write the page at 0xa000 that contains the main function.
- ✓ What we wrote on flash stays even if it's invalid.



## Impact

- This vulnerability impact all devices of Silicon Labs EFM32/EFR32.
- Can be used on any devices as long as you can trigger the OTA.
- No authentication (by default).



**CVE-2023-4041 Detail**

**Description**

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), Out-of-bounds Write, Download of Code Without Integrity Check vulnerability in Silicon Labs Gecko Bootloader on ARM (Firmware Update File Parser modules) allows Code Injection, Authentication Bypass.This issue affects "Standalone" and "Application" versions of Gecko Bootloader.

**Severity** CVSS Version 3.x | CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

**CNA:** Silicon Labs **Base Score:** 9.8 CRITICAL **Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

## Impact

- This vulnerability
Labs EFM32/EFR32
- Can be used on a
trigger the OTA.
- No authenticatior

Finding a vuln in the OTA layer
Finding a vuln in the physical layer

wnload of Code Without Integrity Check
llows Code Injection, Authentication

N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

## What, Why and How?

**RAIL provides a radio interface for different wireless protocols.**

- Handle the low-level operations of the radio peripherals.
- Unlike the rest of the code, it is closed source.
- Obfuscated (symbol names are replaced by `RAILINT_{RANDOM_MD5}`).

**Why?**

- It is a key component of Silicon Labs products.
- There is no alternative to RAIL.
- A vulnerability in this layer will bypass mitigations of the above layer.

**How?**

- Static analysis: Reversing, Scraping data, Leaks...
- Dynamic Analysis: Attach a debugger, Test for specific packets...

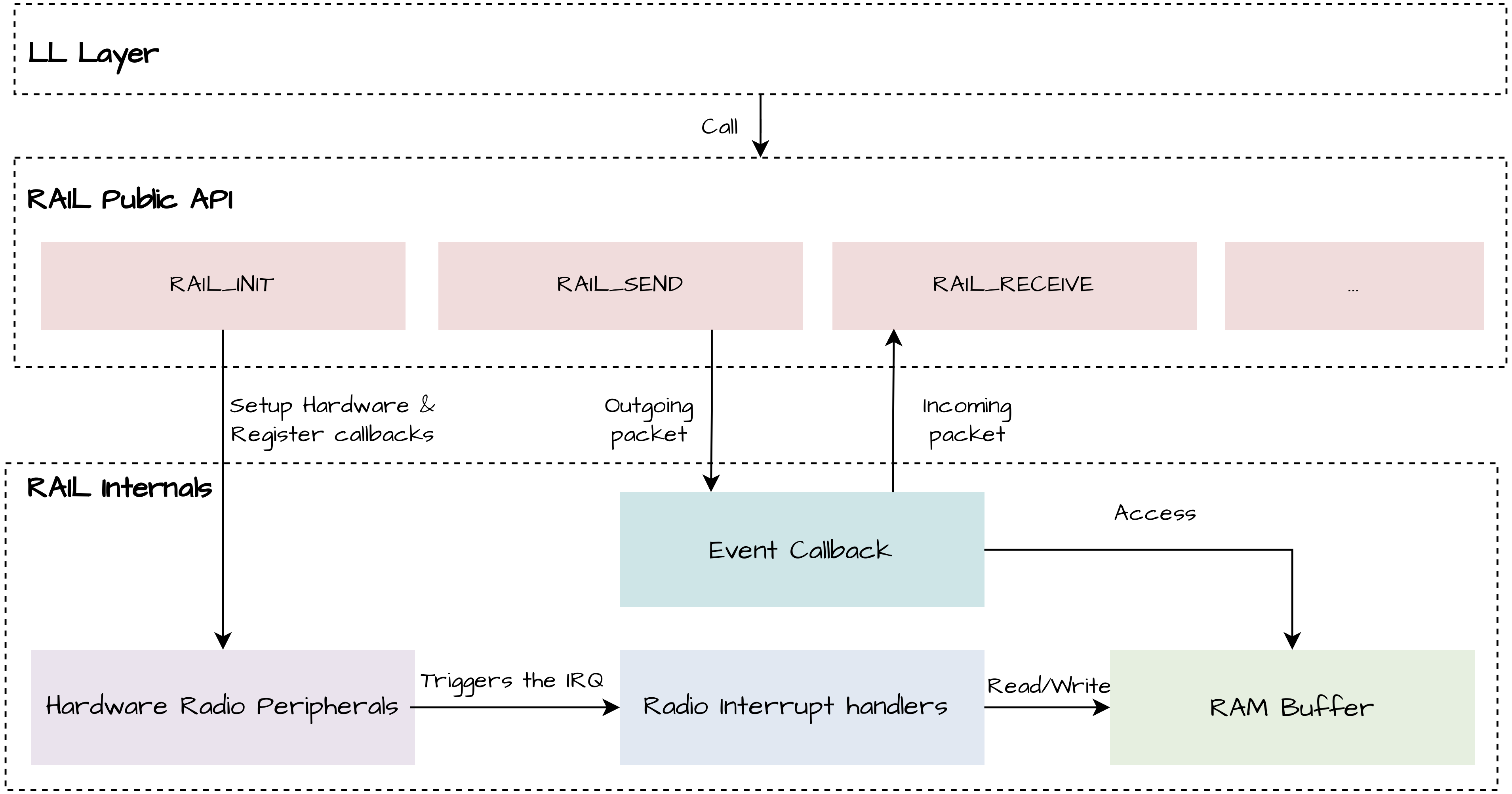
# Static Analysis

```
void RAILINT_3e587e95c937431090ed1e304e004f0f(void) {
    if ((((_DAT_0fe0814c != -1) && (-1 < _DAT_0fe0814c << 1)) &&
    (1 < RAILINT_125ec31d46a455a6cd0914d6eaef7418)) &&
    (RAILINT_125ec31d46a455a6cd0914d6eaef7418 != 8)) {
        RAILINT_99f0e717b1ae5f07aeace88464176fa9(3);
        _DAT_b802104c = 0x4000000;
        _DAT_b8021014 = 1;
        _DAT_b8021028 = 2;
    }
    return;
}
```

# Static Analysis

```
void RAILINT_3e587e95c937431090ed1e304e004f0f(void) {
    if ((((_DAT_0fe0814c != -1) && (-1 < _DAT_0fe0814c << 1)) &&
    (1 < RAILINT_125ec31d46a455a6cd0914d6eaef7418)) &&
    (RAILINT_125ec31d46a455a6cd0914d6eaef7418 != 8)) {
        RAILINT_99f0e717b1ae5f07aeace88464176fa9(3);
        RAC_NS.SR3_SET    = 0x4000000;
        RAC_NS.CTRL_SET   = 1;
        RAC_NS.SEQIF_SET = 2;
    }
    return;
}
```

# Architecture of RAIL



## Dynamic Analysis

We need to be able to:

- Send some malformed packets to see if they are correctly handled.
- Attach a debugger to the chip to trace instructions and dump memory.

## Dynamic Analysis

We need to be able to:

- Send some malformed packets to see if they are correctly handled.

**Problem N°1**: Not possible due to hardware limitations.

- Attach a debugger to the chip to trace instructions and dump memory.

## Dynamic Analysis

We need to be able to:

- Send some malformed packets to see if they are correctly handled.

**Problem N°1**: Not possible due to hardware limitations.

- Attach a debugger to the chip to trace instructions and dump memory.

**Problem N°2**: Not possible because it slows down the chip.

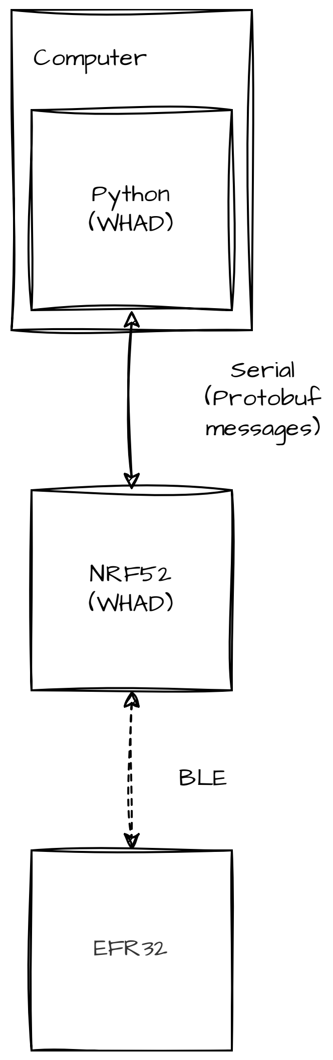
## Dynamic Analysis

## Solution N°1: WHAD For The Win

WHAD is a framework developed by **Damien Cauquil** and **Romain Cayre**.

- Supports wireless protocols such as BLE or Zigbee.
- Supports different hardware: HCI device, NRF52, etc;
- A lot of features: sniffing, hijacking, etc;

WHAD can be used to send a malformed packet and check how the chip is handling it.



## Solution N°2: Developing a DBI

A Dynamic Binary Instrumentation (DBI) injects the instrumentation code directly inside the debuggee which increases the performance.

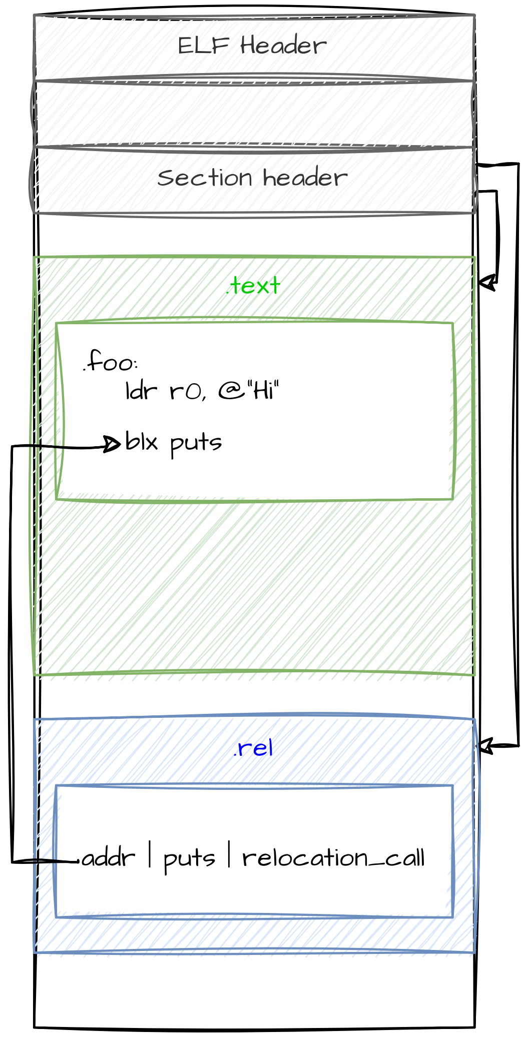
Why do we need it?

- Insert debug stub in critical place where we can not insert a debugger;
- Insert Hooks to change the behavior of some functions.

## How is the DBI working?

How is the DBI working?

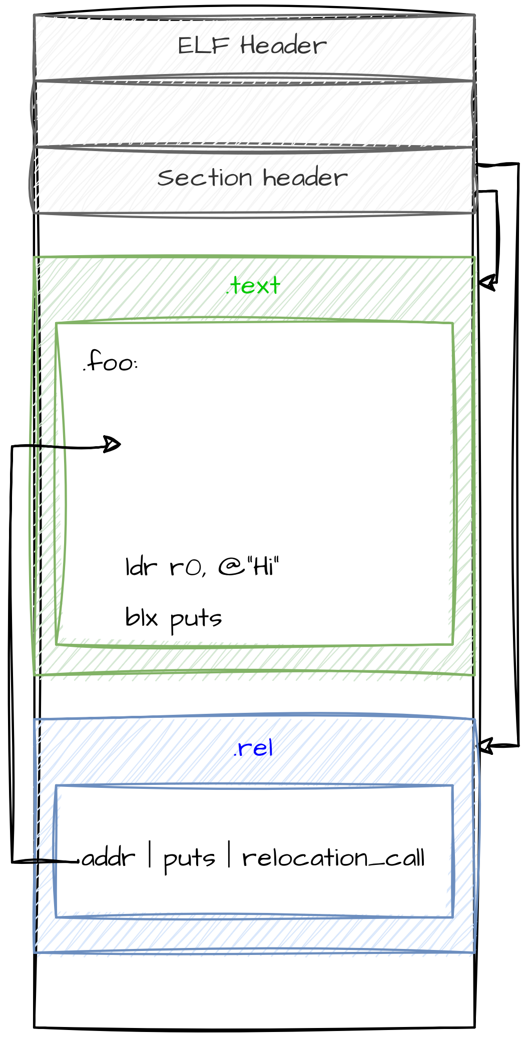
- Convert the Assembly code we want to inject to machine code using `keystone`;
- Extract object files (ELFs) from RAIL library;
- Modify the object files using `LIEF`;
- Repack the library;
- Compile the project with the modified library.



# How is the DBI working?

How is the DBI working?

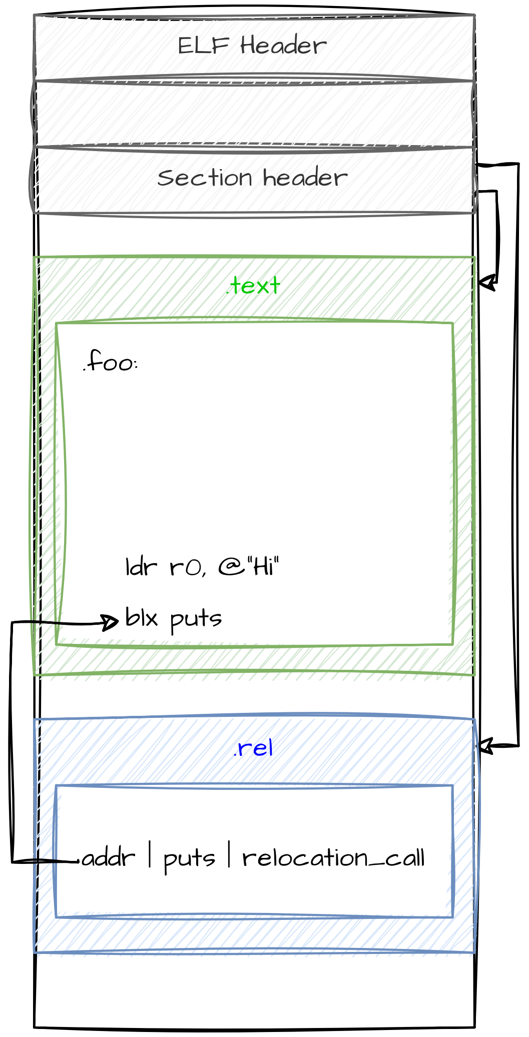
- Convert the Assembly code we want to inject to machine code using `keystone`;
- Extract object files (ELFs) from RAIL library;
- Modify the object files using `LIEF`;
- Repack the library;
- Compile the project with the modified library.



## How is the DBI working?

How is the DBI working?

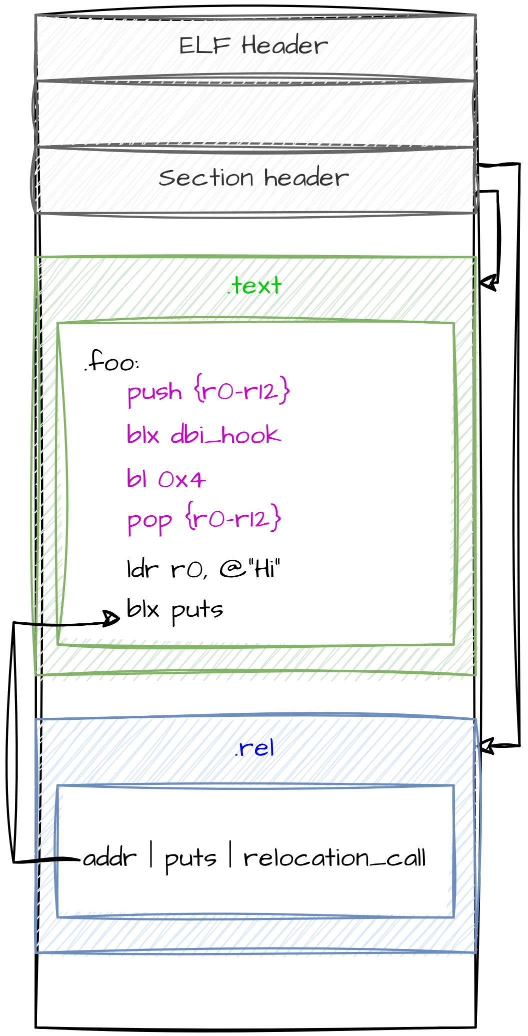
- Convert the Assembly code we want to inject to machine code using `keystone`;
- Extract object files (ELFs) from RAIL library;
- Modify the object files using `LIEF`;
- Repack the library;
- Compile the project with the modified library.



# How is the DBI working?

How is the DBI working?

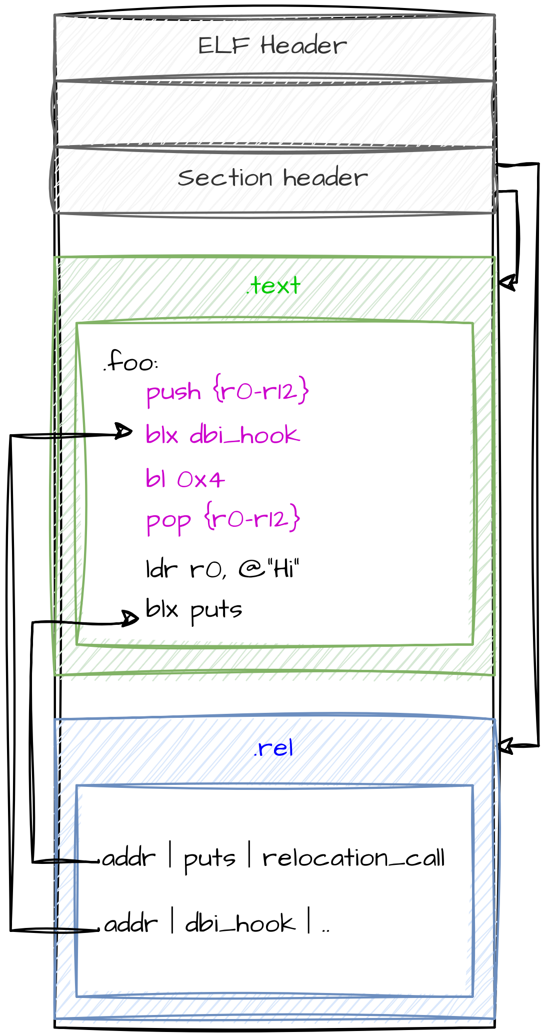
- Convert the Assembly code we want to inject to machine code using `keystone`;
- Extract object files (ELFs) from RAIL library;
- Modify the object files using `LIEF`;
- Repack the library;
- Compile the project with the modified library.



# How is the DBI working?

How is the DBI working?

- Convert the Assembly code we want to inject to machine code using `keystone`;
- Extract object files (ELFs) from RAIL library;
- Modify the object files using `LIEF`;
- Repack the library;
- Compile the project with the modified library.



## Demo Time

## Results

## Conclusion & Takeaway

- Demonstrate how we bypassed secure boot using a classic vulnerability (some vulnerability don't die, they move to better host).
- Fuzzing is not that hard!
- New approach for the DBI.
- Check out the blogpost for more details.
- https://blog.quarkslab.com/breaking-secure-boot-on-the-silicon-labs-gecko-platform.html

# Thank you!

Contact information:

|  |  |
|---|---|
| Email: | bforgette@quarkslab.com |
| Twitter: | https://twitter.com/Mad5quirrel |
| Phone: | +33 1 58 30 81 51 |
| Website: | https://www.quarkslab.com |

Quarkslab