Secure Firmware Distrubution for Automotive Control

struct by_lightning{};

This technical documentation is our design for the BWSI Embedded Security and Hardware Hacking Design Challenge.

Table of Contents

I. Introduction

A. Abstract

Struct by_lightning's innovation is an embedded system that supports secure firmware distribution for self-driving automotive control. This system is a crucial factor in supporting the development of self-driving automobiles, contributing to the news evolution in transportation and automobile safety.

Struct by_lightning is designed with firmware and bootloader updates and respective tools to encrypt firmware and provide safe versioning updates. In addition to the encryption of the metadata and frame structures; these procedures help monitor system performance and provide the transfer of data. *Struct by_lightning* is equipped with numerous algorithms and tools capable of defending against Man in the Middle (MITM) attacks, buffer overflows, modified cryptographic signature attacks, and flash memory read attacks.

This technical documentation describes the design and development process that makes *Struct by_lightning's* embedded system the most secure to assist MITRE in their development of a safe and secure self-driving car.

Fig. 1
Struct by_lightning team members

II. Design Rationale

A. Design Evolution

Struct by_lightning focuses on three design principles: simplicity, security, and reliability. In one of our first design meetings we listed the MITRE challenge requirements and reflected on different design ideas and possible code implementations. In our analysis of these components, we created different flowcharts and designs of reliable and secure code that performed core bootloader and firmware system requirements.

Struct by_lightning implements variations of simple algorithms that we have learned in the Beaver Works Summer Institute (BWSI) Embedded Security and Hardware Hacking course, such as Keyed-Hashing for Message Authentication (HMAC) and the Advanced Encryption Standard (AES) symmetric block cipher. These cryptographic algorithms allow *struct by_lightning* to not only protect but secure data packets, metadata, firmware, and various tools.

B. Flowchart

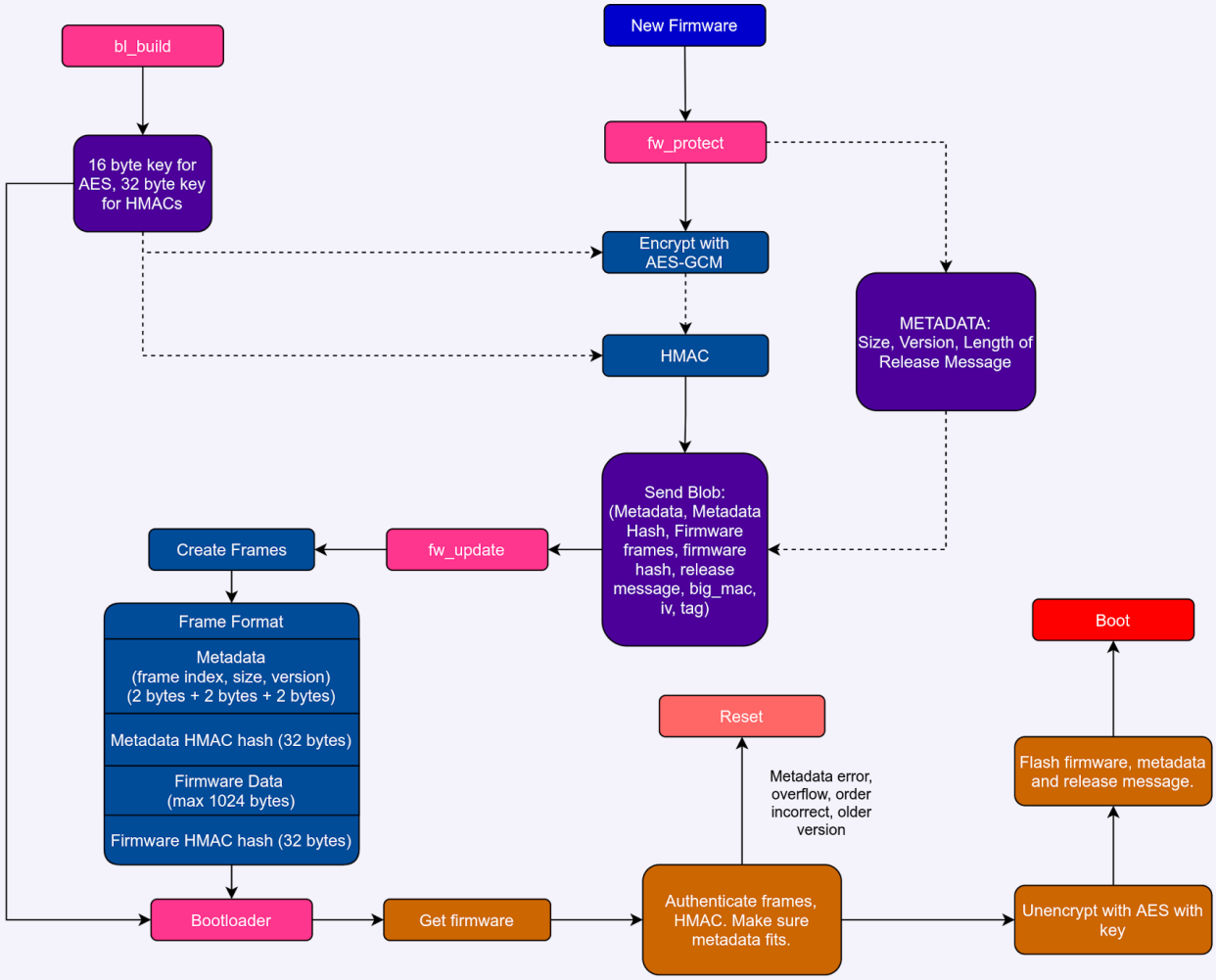

Fig. 2
Flowchart of design

C. bootloader.c

The bootloader reads in AES and HMAC keys from `secret.h`. The bootloader then uses this cryptographic information to decrypt and verify data using AES-GCM and HMAC SHA-256. This verification allows an extra procedure step to verify that packets have been sent accurately and in the correct order.

Here are the main functions that were modified for our bootloader:

```
void load_firmware(void);
int gcm_decrypt_and_verify(char* ct, int ct_len);
int sha_hmac(char* data, int len);
```

Fig. 3
Main functions modified in the bootloader

We added in our own wrappers for AES decryption and HMAC verification. The core code of these functions was taken from `beaverssl.h`, but we simplified them for our use. The inputs, the keys, hashes, IVs and tags, are all self-contained within the function which helps organization. In addition, the HMAC verification is done in constant time, so it will fend off timing attacks.

The `load_firmware()` function, which is the bulk of the code, has a long process to flash firmware. It first goes into update mode and receives firmware metadata along with the first of many HMAC hashes. All firmware and frame metadata variables are checked against hard limits after verification. All `for` loops have similar hard limits to protect against overflow. The bootloaders give the OK to read in frame metadata, 1024 byte frames and each of their hashes. Each metadata contains version, size and index. The versions are compared with the firmware metadata and the indices are tested against the bootloader's counter. More detail on the order and size of data and protection is in the security measures section and in the `bootloader.c` file. This focuses more on bootloader specific tasks. Once all firmware is read in, it verifies the entire firmware. The bootloader reads and verifies the release message using a length from the firmware metadata. After, it verifies the entire firmware, firmware metadata and release message together. All verification is done with HMAC-SHA256. The data is then decrypted with our AES-GCM function. The firmware is flashed in 1024 byte page sizes. The metadata and release

message are flashed in their own separate pages. This is because the release message has a max of 1024 bytes, so it will need an entire page at its maximum size.

The release message is stored differently than in the example bootloader. We modified the `load_initial_firmware()` to flash the message in its new location. The release message address never changes and its length is kept in the flashed metadata. The `boot_firmware()` function uses the length with a hard limit of 1024 bytes to print out the release message instead of `uart_write_str()`. This ensures it will not output more than it should.

We created other simple functions to help with organization. The `send_err()` function makes it easy to send an error message, send an error to `fw_update` and reset the system. Similarly, `check_read()` checks the read variable and resets accordingly while `uart_read_variable()` reads in data of variable length from UART.

a. Tools

i. bl_build.py

When bl_build.py is run, the program generates a 16-byte AES-128-GCM key and a 32-byte HMAC-SHA256 key. These keys are written to both secret_build_output.txt in the /tools folder and to the secrets.h file in the same folder as `bootloader.c`.

ii. fw_protect.py

The `fw_protect.py` pool reads the AES and HMAC keys from the `secret_build_output.txt`, and creates the blob to send to `fw_update.py`. The structure of the blob is as follows:

```
Metadata
Metadata Hash
Firmware Data
Release Message Data
Big_Mac
IV+TAG
```

Fig. 4
Data blob sent to `fw_update.py`

The Metadata consists of 6 bytes: 2 bytes for the version number and 2 bytes for the length of the unencrypted firmware and 2 bytes for the length of the release message.

The Metadata Hash is a 32-byte HMAC-SHA256 hash of the Metadata using the HMAC key generated during `bl_build.py`.

The Firmware data consists of a looped page format: page metadata, page metadata hash, page data, page data hash. The page metadata has 2 bytes for the page index (so order of pages does not get misarranged), 2 bytes for the page size, and 2 bytes for the version number. The page metadata hash is a 32-byte HMAC-SHA256 hash of the page metadata. The page data is the encrypted firmware data, cut into slices with a maximum length of 1024 bytes per page. And finally, the page data hash is a 32-byte HMAC-SHA256 hash of the firmware data. This [2][2][2][32][<=1024][32] byte frame is looped and appended to the total blob per iteration. At the end of the loop, a 32-byte HMAC-SHA256 hash of the entire encrypted firmware data is appended to the blob.

The Release Message Data consists of two elements: the release message itself and a 32-byte HMAC-SHA256 hash of the release message.

The Big_Mac is a 32-byte HMAC-SHA256 hash of three combined elements: the entire encrypted firmware, the Metadata (from above), and the release message. This is here for an extra layer of integrity and authenticity checks.

The IV and TAG of the AES-GCM encryption are also appended to the blob, respectively (both 16 bytes).

This large blob is sent over to `fw_update.py`, where it will be sliced and sent to the bootloader in chunks.

iii. fw_update.py

The fw_update tool receives the blob from `fw_protect.py` and slices the first 38 bytes of the blob into a metadata chunk, leaving the rest as a `data_blob`. The metabyte chunk consists of: 2 bytes for the version, 2 bytes for the length of the unencrypted firmware, 2 bytes for the length of the release message, and 32 bytes for the metadata hash. This metadata chunk is then sent to the bootloader over the serial connection. The program then waits for an OK message from the bootloader.

To cycle through the frames, a loop first checks the metadata of the current frame by slicing bytes from the beginning of the `data_blob`. Once the length of the index of the frame, size of the current encrypted firmware page, version number, encrypted firmware page, and HMAC hash is determined, the frame is written to the bootloader over serial and the frame is popped from `data_blob`. The program waits for an OK message before looping again. This process loops until there are no more pages to send (the number of pages is known because of the initial metadata). The program then waits for an OK message from the bootloader.

What is left of the `data_blob` is the hash for the entire encrypted firmware, the release message data, big_mac, iv, and tag. The 32-byte HMAC hash of the entire encrypted firmware is first sent to the bootloader over the serial connection. After waiting for an OK message, the release message and its respective hash is sent to the bootloader, and after waiting again, the big_mac is then sent to the bootloader.

Finally, after one last OK message, the iv and tag are sent over to the bootloader over the serial connection, and the program sends a zero length payload to tell the bootloader to finish writing its page. The length of each of these chunks are predetermined and sliced from `data_blob`.

III. Security Measures

A. Security Highlights

In *struct by_lightning's* embedded system, we have numerous systems and procedures implemented to maximize the security and the performance of our bootloader.

Our biggest form of security is the use of cryptographic algorithms likc HMAC SHA-256 Encryption and AES-128 GCM Symmetric Block Ciphers. The below block showcases what HMAC hashed.

```
(x) denotes x bytes
=============================================================================================
       (2) version
       (2) total firmware size
       (2) size of release message
(32) hmac hash of the above 6 bytes (metadata hash)
=============================================================================================
       LOOPED: {
              (2) frame index
              (2) frame size
              (2) version
              (32) frame metadata hmac hash (hash of frame metadata)
              (frame size) encrypted frame data page
              (32) frame data hmac hash (hash of frame data + frame metadata)
              }
(32) hmac hash of total firmware data
=============================================================================================
{      (variable) release message       }
(32) hmac of release message
=============================================================================================
(32) hmac of (total encrypted firmware + entire firmware metadata + release message)
=============================================================================================
```

Fig. 5
Diagram of different HMACs used

By using HMACs on both metadata and individual data packets, *struct by_lightning* successfully protects against integrity and authenticity attacks. This also prevents MITM attacks, where teams would be able to disrupt the order and flow of our packets and data itself.

AES GCM is also showcased in the block above, where frame data is encrypted. This cryptographic block cipher provides both data authenticity and confidentiality through a 128-bit cryptographic seal. The firmware is decrypted after all the HMACs are verified.

To prevent attacks, the *struct by_lightning* embedded system attempts to immediately cease updating at any sign of disruption. A myriad of HMACs used throughout our encryption and decryption should sufficiently protect against integrity and authenticity attacks. Numerous checks during bootloader.c for inputs with incorrect size, version, and indexes should prevent against version rollback attacks. Finally, our AES-GCM encryption should sufficiently provide confidentiality for our system.

B. Cryptographic Algorithms

a. AES-128 GCM Symmetric Block Cipher

The AES-128 GCM Symmetric Block Cipher is a fast symmetric-key block cipher. This encryption is fast and secure, and provides authentication for packet data transfers.

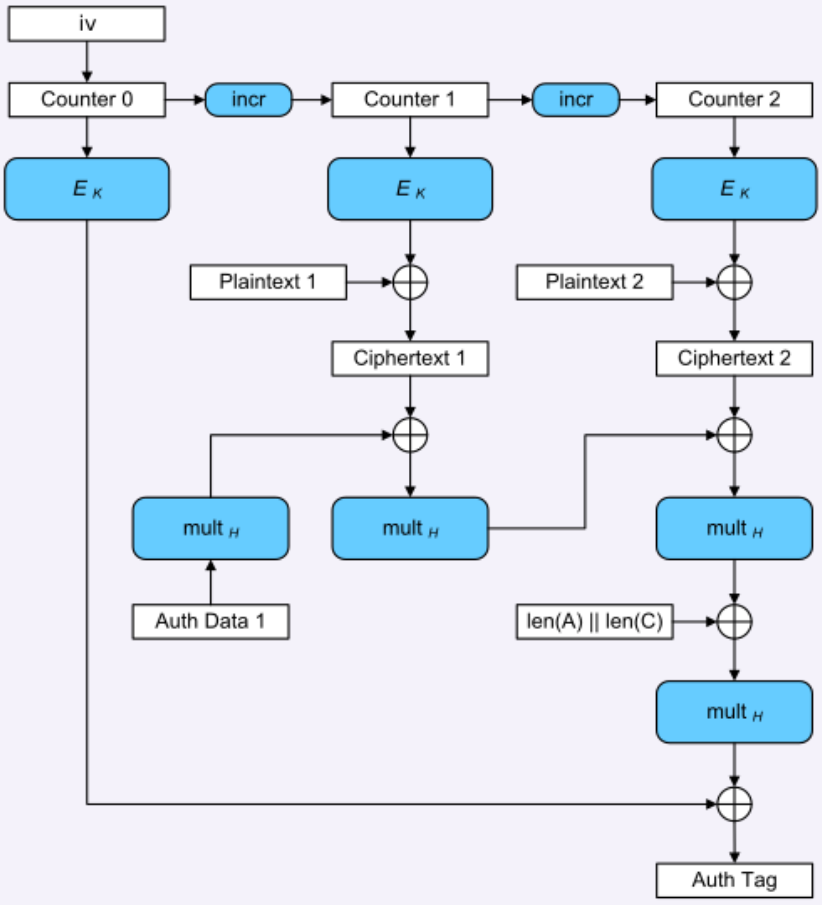

Fig. 6
Diagram of AES-GCM encryption

b. HMAC SHA-256 Encryption

HMAC is a type of method authentication code which provides authentication with a shared key. Since keys are completely inaccessible to attackers in this design challenge, this method of authentication was chosen due to its speed and effectiveness.

IV. Conclusion

A. Closing

Struct by_lightning has designed an embedded system that has not only met MITRE's proposed requirements for the 2021 Design Challenge, but has exceeded and further protected against numerous exploits such as Man in the Middle (MITM) attacks, buffer overflows, versioning exploits, modified cryptographic signature attacks, and flash memory read attacks.

Using our time efficiently, *struct by_lightning* was able to brainstorm and work together as a team to plan, develop, and test code for our firmware and bootloader. Our teamwork was not only fast, but efficient, with each member taking different sections of programming and documentation to help increase work productivity and to finally develop a running and booting prototype and final design.