scriptohio

The Ohio State University

Andrew Haberlandt, Yu-Shiang Jeng, Youssef Moosa, Kyle Westhaus
Advised by: Dr. Rajiv Ramnath, Aaron McCanty

Design Overview

- Symmetric key for encryption, asymmetric key pair for auth
- Keys stored in EEPROM
- Flash verification after flash write/erase

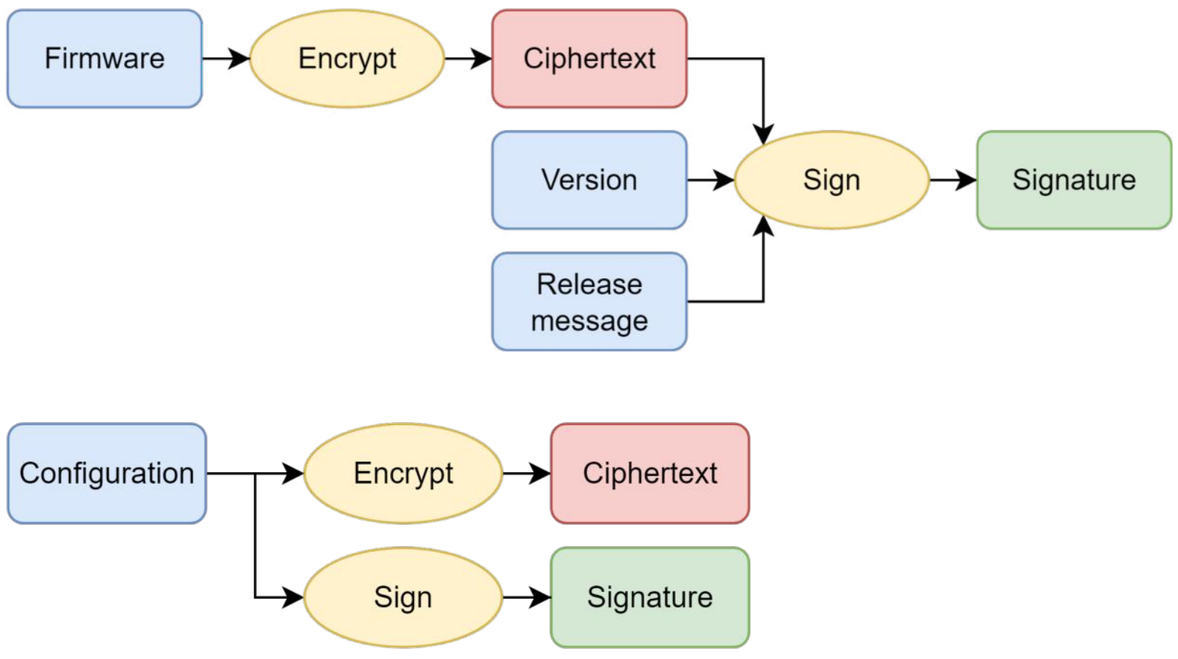


Figure 1: Encrypt-then-sign for firmware, encrypt-and-sign for configuration

Defensive Highlight

Trojan Defense	
Verification	Hash and verify flash on every flash write and erase
Isolation	All verification code placed in SRAM
Diversification	Build-to-build diversity: random NOPS to place functions at different locations

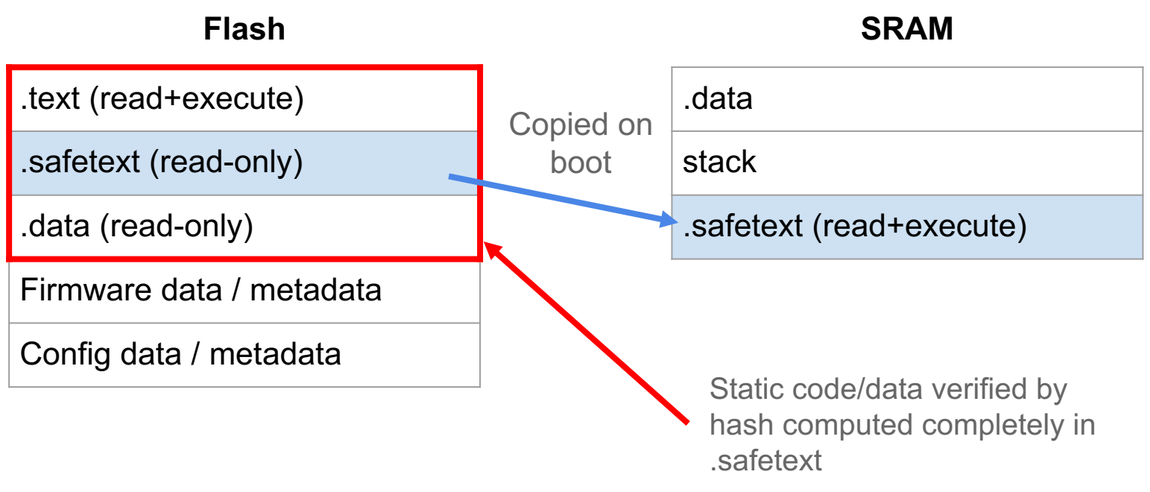


Figure 2: Using SRAM functions to verify Flash regions

Mitigations: stack canaries, auto zero-init, custom stack ASLR, memory protections, EEPROM block hiding

What would we do differently?

- Do not allow attackers to write controlled data to flash
- Wipe keys in EEPROM when tampering detected
- Different keys for firmware and configuration

Offensive Highlight

Trojan attack: Trigger string

- When a specific flash sequence of bytes (the trigger string) is written to flash, write a shellcode stub to `0x5800` (address of `Bootloader_Startup`)
- Our shellcode stub can then be executed by resetting the device at any time
- Vulnerability: Reading from UART (which an attacker can hang) between unsafe flash writes and verification
- Fix: No blocking operations between writes and verification

Listing 1: Trojan source code

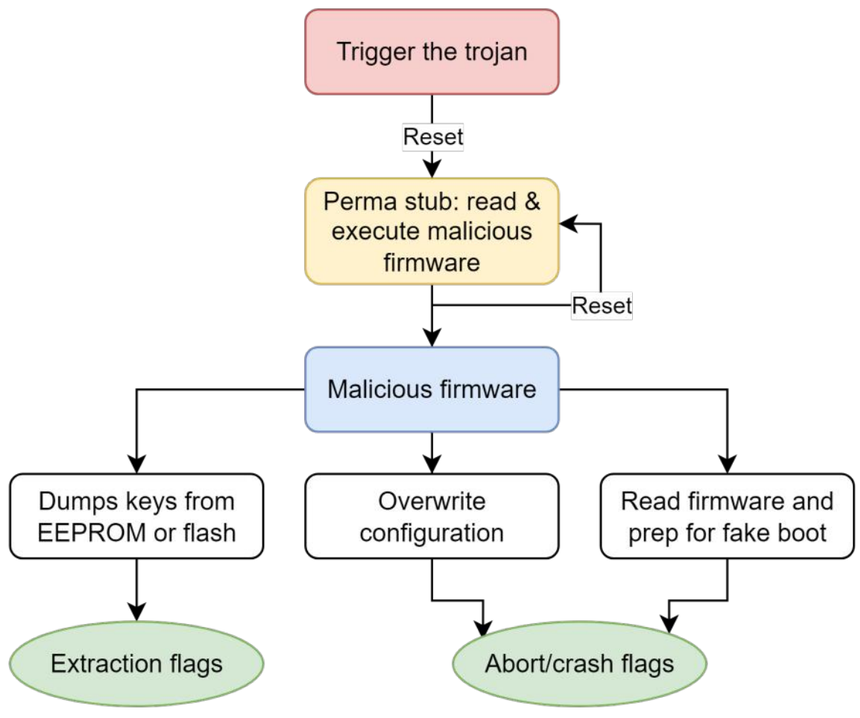
```
uint8_t
flash_trojan(uint32_t fma, uint32_t fmd, uint32_t fmc, uint8_t operation) {
    if (operation == 0 && !triggered) {
        int status = (fmd == trigger_data[trigger_idx++]);
        if (!status)
            trigger_idx = 0;
        else if (trigger_idx == TRIGGER_WORD_SIZE) {
            int i;
            trojan_flash_erase(TARGET_ADDR);
            for (i = 0; i < SHELLCODE_SIZE; i++)
                trojan_flash_write(TARGET_ADDR + 4 * i, shellcode[i]);
            triggered = 1;
        }
    } else if (
        triggered && fma >= TARGET_ADDR && fma < TARGET_ADDR + FLASH_PAGE_SIZE)
        return 1;  // Do not perform the operation if it targets our shellcode
    return 0;      // perform operation anyway
}
```

Listing 2: Code vulnerable to our trojan

```
void handle_update(void) {
...
    // Receive release message
    rel_msg_size = uart_readline(HOST_UART, rel_msg + 8, 1025) + 1; // Include terminator
...
    handle_update_write(rel_msg, ...);
}

void handle_update_write(uint8_t* rel_msg, ...) {
...
    flash_write_unsafe((uint32_t *)rel_msg_read_ptr, rel_msg_write_ptr, rem_bytes >> 2);
...
    // Retrieve firmware
    load_data_unsafe(HOST_UART, FIRMWARE_STORAGE_PTR, size, FIRMWARE_MAX_SIZE);

    current_hash(hash2, (uint8_t*)FIRMWARE_BASE_PTR, padded_size);
}
```



Hi we are team <team name> and our bootloader was a modified version of the C insecure example provided my MITRE. To meet the security requirements of the competition, we add a symmetric key to the bootloader for encryption, and an asymmetric key pair used for authenticating firmware/configs between the host and the device. We use XChaCha20 to protect the confidentiality of the firmware and configuration, preventing attackers from extracting IP from such files. To ensure integrity and authenticity of the firmware and configuration files, we use EdDSA to sign each protected file. By storing only the public key in the bootloader to perform verification, we ensure that attackers can never forge a new signature for a malicious firmware or configuration. Finally, to address the flash trojan, we implemented verification of flash after every write and erase, which we will talk about more later.

There were a number of defensive measures which we found worked particularly well. We were able to enable a number of compiler mitigations, including stack canaries and variable auto zero-init. Stack canaries make it more difficult to overwrite the return address in the case of a memory corruption bug which might otherwise lead to arbitrary code execution and complete compromise of the device. We noticed memory corruption issues in other team's bootloaders which could have been prevented with stack canaries, so we consider this to be a highly successful mitigation. Variable auto zero-init prevents garbage stack data (potentially containing secrets) from making its way into program variables, but we have no evidence that this was necessary due to the lack of memory corruption bugs in our design. We also implemented our own stack ASLR, based on our custom pseudo-random entropy source. This was supposed to make it more difficult to launch a stack-based exploit, since stack addresses would not be predictable. We have do not believe any teams succeeded in any stack-based exploit on our design, so we believe this was effective. Lastly, we enabled the Memory Protection Unit, and configured each region to the minimum permissions necessary. This allowed us to ensure that non-code segments are not executable. We think this was effective, since in other team's designs we noticed that in the event of a memory corruption bug, we could jump to shellcode anywhere which made exploitation much easier.

One mitigation we want to talk more about is our primary trojan mitigation: flash verification. Our design hashes and verifies all of the static portions of flash, shown in red in Figure 2, after every flash write and erase. We store our flash verification code in a special segment we call "safetext". This area of memory is copied from flash to SRAM on boot and contains everything needed to hash the static regions of flash, and compare the resulting hash with the hash stored in EEPROM at build-time. Having this verification code execute in SRAM prevents the flash trojan from overwriting the code that is doing the verification. This required us to ensure that every instruction that was used for hashing or EEPROM reading actually resided in SRAM – otherwise, the mitigation would be pointless. We used binary ninja, a reverse engineering tool, to audit our compiled bootloader to ensure that the verification code never calls code not in SRAM. The final piece of our trojan mitigation is "diversification" – we wanted to ensure that every time the team reprovisions our device, they get a slightly different bootloader out as a result. This would make it harder to hard-code patch addresses into the trojan.

When verification of flash failed, we chose to erase flash at 0x5800, which is permitted in the competition because the only way verification fails is if the flash trojan activates. We think a much better mitigation would be to wipe EEPROM keys, since this operation cannot be trivially blocked by the flash trojan. Other things we might have done differently would be to use different keys for firmware and configuration. To make developing the trojan even harder, we would want to eliminate flash writes of un-authenticated data. This is because even if we will refuse to boot, writing controlled values makes it easy to write a trojan that triggers upon writes of specific values.

After focusing on trojan defenses during the design phase, we had the chance to develop a flash trojan as part of the attack phase. As seen in Listing 1, our flash trojan intercepts all writes (operation 0) to flash memory. If the trojan has not already been triggered, it waits for a group of 16 trigger bytes (sent by the attacker) to be written before activating. Upon activation the trojan will erase the flash page starting at 0x5800 (the bootloader entrypoint) then write shellcode there and record that it has been triggered. For writes after the trojan has been triggered, the trojan will inspect the destination address and reject anything which would overwrite the shellcode page. Since this trojan writes shellcode to the bootloader entrypoint, a simple soft or hard reset after triggering allowed us to obtain code execution and compromise all security guarantees of the device.

Note that this trojan modifies the bytes in flash and thus is detectable by any design which hashes the contents of flash before and after write operations. In order to work around this, we had to find times during the execution of each design where we could cause a reset before any destructive action occurred. While we were able to obtain this primitive on all teams we used the trojan for, we would like to highlight one specific attack we performed against a team. Relevant sections of their code are shown in Listing 2.

For our attack on this design, the attacker-controlled bytes to be used for the trigger are sent as the release message. The release message is read from UART then passed to handle_update_write(), where it is written to flash. While there is a current_hash() function which would verify the contents of flash and panic after the trojan triggers, the vulnerability is that there is a call to load_data_unsafe() between the write and the verification. load_data_unsafe() performs a blocking read on UART, which we leveraged to perform a reset and reach our shellcode. We propose moving the unsafe write of the release message to after the call to load_data_unsafe() and immediately before current_hash(). This change would have forced us to beat a race condition with verification and would have been significantly harder to exploit.

Once the trojan was triggered, the shellcode allowed us to upload pre-compiled malicious firmware to perform various tasks useful to us as attackers. This includes dumping keys from EEPROM or flash to decrypt protected firmwares and configurations. We can also overwrite existing configurations in flash, which is useful for causing a flight abort. Our malicious firmware also allows us to send a different firmware and respond with fake responses to convince the "boot" host tool to boot into the new firmware. This was useful for obtaining both flight abort and aircraft crash flags.