UDC 621.396.67 *http://dx.doi.org/10.30970/vam.2024.32.13307*

MITIGATING SECURE BOOTLOADER VULNERABILITIES IN "FLASHLESS" MICROCONTROLLERS

M. Shcherbyna, P. Venherskyi

Ivan Franko National University of Lviv,
1, Universytetska str., 79000, Lviv, Ukraine
e-mail: mykola.shcherbyna@lnu.edu.ua, petro.venherskyy@lnu.edu.ua

The potential for man-in-the-middle (MITM) attacks targeting secure bootloaders in microcontrollers without internal flash memory is examined, utilizing a device that monitors SPI bus communication. The possibility of bypassing cryptographic signature verification of embedded firmware is demonstrated by precisely identifying the optimal moment for fault injection or executing modified firmware immediately after signature validation. Additionally, the limited ability to modify AES-CTR encrypted code without knowledge of the encryption key is illustrated. A hardware-based protection mechanism is proposed to mitigate the described MITM attack during execution by employing message authentication codes.

Key words: microcontroller, flash memory, secure bootloader, MITM, digital signature, message authentication codes, AES-CTR, decryption.

1. Introduction

Over the past five years, microcontrollers (MCUs) without internal flash memory have appeared on the market. Notable examples include the RP2040/RP2350 from Raspberry Pi [1], the LPC18S50/S30/S10 series from NXP [2], and the CYW20829 from Infineon Technologies [3]. In these "flashless" architectures, firmware resides on an external flash memory chip, such as the W25Q128JW from Winbond [4], which communicates via SPI or QSPI interfaces (Fig. 1). This represents a return to the foundational principles of MCU design, as early models relied on external ROM, contrasting with the recent trend of maximizing internal flash memory capacities [5]. A specialized QSPI peripheral module facilitates the seamless integration of external flash memory into a predefined address range designated for XIP (Execute in Place) functionality. Enhanced by an internal cache, the module dynamically retrieves data using SPI Fast Read commands to get bytes on the fly.

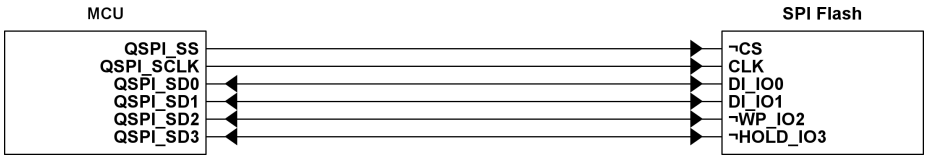

Fig. 1. Typical embedded system with flashless MCU and external SPI Flash

At the same time, cybersecurity concerns in embedded systems are becoming increasingly relevant. Cryptographic verification of firmware signatures is becoming an industry standard. For external flash memory, encryption is of utmost importance. The firmware is decrypted on the fly by the aforementioned "XIP" peripheral block. The separation

of the MCU and flash provides flexibility in selecting the specific external memory IC model during the preproduction phase, based on the final firmware size and data storage requirements. However, it remains unclear whether this separation has introduced new potential vulnerabilities. Resolving this uncertainty is the main goal of our research.

2. Hardware Modifications

Let us assume that we have placed our "magic" device in the communication line between the microcontroller and the external flash memory (Fig. 2). From the perspective of the MCU, it functions as a SPI Flash, and from the perspective of the flash memory, it operates as a microcontroller.

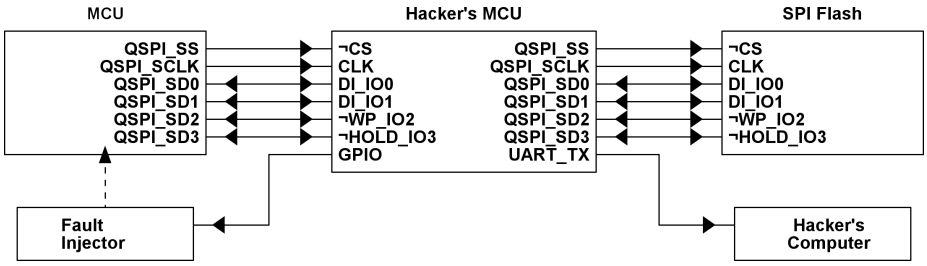

Fig. 2. Modified embedded system with flashless MCU and external SPI Flash

Furthermore, consider that the MCU in the system under investigation can be subjected to fault-injection attacks [6] (voltage glitching, clock glitching, or EMI), triggered by our device via its GPIO pin. Lastly, our device is capable of transmitting logs to a PC through a UART interface. Note that in the attacks described below, we will not necessarily utilize all the capabilities of this universal system.

3. Secure Boot Process

The purpose of secure bootloaders, whether BootROM or Stage1, is to verify the digital signature of the Stage1 bootloader and firmware, respectively. For the purposes of this discussion, we will assume that we are working with the Stage1 bootloader, which verifies and launches the firmware. Generally, the following steps are performed:

1) Read a "header" structure to determine the location of the firmware and its signature.
2) Compute the firmware hash (this involves a complete read of the firmware).
3) Read the firmware signature.
4) Decrypt the signature and compare the hashes.
5) If the hashes match, transfer control to the firmware.

Steps 1) and 3) are sometimes merged. Steps 2) and 3) can be interchanged. Additionally, step 2) is typically performed block-by-block, sometimes explicitly, and sometimes due to the limited cache size of the MCU's XIP peripheral.

Assume the magic device (Fig. 2) observes (Q)SPI read commands $R_i(a_i, n_i)$, which return the requested byte sequences B_i, where $|B_i| = n_i$, and $0 \leq i < m$. Here, $a_i \in A$ represents an address within the flash memory's address space A, and n_i denotes the corresponding size. The header and signature are represented as B_h and B_s, respectively, while other sequences are part of the firmware F.

The computed hash is $H(F) = H(B_{j_1} \| B_{j_2} \| \dots \| Bj_k)$, where $j_i \neq s$ (B_h may or may not be part of the signature). The stored hash is $E_{Kpriv}(H(F))$, where (K_{pub}, K_{priv}) is a key pair used for signature verification. The firmware is considered authentic, if $H(F) = D_{Kpub}(B_s)$.

If the firmware is encrypted with the key K_{sec} using an algorithm such as AES-CTR, which allows random access, the following relationship holds true: $F_D = F_E \oplus X$, where X is the keystream generated from K_{sec}.

4. Security Threats to Embedded Systems

When physical access to a device is obtained, typical objectives of attackers include:

1) Extracting the device's firmware F for subsequent static analysis (reverse engineering);
2) Executing a modified (patched) firmware F_P for dynamic analysis.
3) Deploying custom firmware to retrieve the code of the Stage1 bootloader and / or BootROM, data in SRAM after the Stage1 bootloader, OTP content, etc.

Ultimately, attackers often aim to steal intellectual property to create clones, discover vulnerabilities to facilitate the "legitimate" loading of custom firmware, or gain unauthorized remote access to the embedded system.

For unencrypted firmware, extracting the device's firmware F is as straightforward as reading the contents of the external SPI Flash using regular flash programmer hardware. For encrypted firmware, it is a challenging task, as described later. To achieve other objectives, an attacker would need to compromise the signature verification mechanism. For example, a "secure" MCUBoot bootloader [7] includes the following code:

```
rc = ED25519_verify(buf, blen, sig, pubkey);
if (rc == 0) {
    /* if verify returns 0, there was an error. */
    FIH_SET(fih_rc, FIH_FAILURE);
    goto out;
}
FIH_SET(fih_rc, FIH_SUCCESS);
```

Despite fault injection hardening, it is technically possible to bypass the **if** (rc == 0) ... condition, effectively marking the firmware as "valid". The only effective countermeasure in this scenario is enabling the *FIH_ENABLE_DELAY* MCUBoot's feature, which introduces a random delay prior to the signature verification process.

In the case of internal flash memory, the MCU functions as a black box, making it challenging to determine the precise moment for fault injection. Open-source bootloader analysis and power consumption measurements can provide some insight, but bypassing the signature often requires days or even weeks of unsuccessful attempts. However, in the external memory scenario, the use of a "magic" device (acting as a sniffer) allows us to almost precisely identify the moments when the signature and/or firmware are read. This significantly reduces the time window required for an attack. This represents the first vulnerability, which, as previously mentioned, can be mitigated by introducing random delays.

The second vulnerability is more critical. By detecting the final pre-verification read, we can pass the original firmware blocks B_i through our "magic" device up until that moment. Subsequently, we switch to sending the patched blocks $\dot{B}_i$, as the signature has

already been verified. This mechanism is hindered only by potential XIP cache issues, which makes the attack nearly always successful. Unfortunately, this vulnerability can only be mitigated through hardware methods.

The actual implementation of QSPI XIP peripherals and their caching algorithms varies across MCU vendors. However, let us assume that the peripheral always reads fixed-size blocks C_i into the cache, where their addresses p_i are aligned with the cache block size n. It is important to note that such XIP caching may result in a situation where, for all read operations $R_i(a_i, n_i)$ described above, $n_i = n$ and $a_i \bmod n = 0$.

This smooths the SPI flash access patterns, making it difficult to distinguish, for instance, B_s from other blocks. In fact, such a block may no longer exist, as the signature could be contained within or distributed across multiple read blocks. Nevertheless, combining reading pattern analysis with inter-read delay measurements still enables the precise identification of the moment preceding signature verification. As a result, the first potential vulnerability described earlier remains present.

The original blocks B_i and (later) the patched blocks $\dot{B}_i$ are transmitted over the SPI bus, each identified by the address a_i. Assume the cache can hold N blocks, each of size n. If the XIP caching algorithm follows a FIFO strategy, the second vulnerability can only be exploited after N *distinct* reads. In the case of an LRU strategy, the calculation becomes more complex. However, due to the bootloader's typical behavior, blocks are rarely requested twice, as B_h and B_s are generally loaded into SRAM after the first read, while firmware blocks are typically accessed only once.

We need extra SRAM of size $r \cdot |A|/n$, accessible by QSPI XIP peripheral, where A is a XIP address space and r is size of MAC code, $r \ll n$. When any block of size n is read during the boot phase, the corresponding MAC is calculated, and its value is stored in that RAM. Then after successful signature verification bootloader turns on the firmware integrity checking that cannot be turned back until reboot, so when any block is read into the XIP cache, its value is checked against the MAC, and a hardware fault is raised in the case of mismatch. Such a solution completely mitigates the second vulnerability.

Using a real MAC code instead of a CRC is essential [8]. A reasonable choice would be a 32-bit UMAC tag [9], with a nonce that is randomly generated upon each reset.

5. Handling Encrypted Firmware

Finally, a compromised signature verification mechanism has little impact if the firmware is encrypted, and the attacker cannot modify it at will. For XIP peripherals, on-the-fly decryption at arbitrary addresses is essential. As noted earlier, AES-CTR is a reasonable choice for this purpose, and there is evidence that one vendor's tools implement this exact algorithm [10].

Although there is no direct exploit, certain techniques can still be leveraged. Suppose the firmware under investigation outputs text information via UART, USB, or another communication interface. Upon startup, it prints: "*Secret firmware v. 1.0*$\mid n$". The referenced message m resides in the *.rodata* section, following the code in the *.text* section, and is present in the encrypted firmware image F_E at a specific address a_m – naturally, in encrypted form (Fig. 3).

We now proceed with the following steps, ideally automating the process. Starting approximately from the middle of the firmware, we advance in increments of 17 bytes (i.e., message length, excluding the terminating null byte). A copy of the encrypted firmware is created, with the byte at the current offset XORed with 1. The altered firmware is then executed using the previously described exploit. If the message is not

a_m+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
m	S	e	c	r	e	t		F	i	r	m	w	a	r	e		v	.	1	.	0		\n	\0
F_D	53	65	63	72	65	74	20	66	69	72	6D	77	61	72	65	20	76	2E	20	31	2E	30	0A	00
= K	84	60	45	CA	7D	2B	D6	B5	08	46	FD	77	5F	B3	43	17	ED	57	AF	76	50	BB	ED	48
⊕ F_E	D7	05	26	B8	18	5F	F6	D3	61	34	90	00	3E	C1	26	37	9B	79	8F	47	7E	8B	E7	48

Fig. 3. Representation of the message in encrypted and decrypted form

received (resulting in a crash) or matches the original output, we proceed to the next iteration. If a different message is received, the process is halted:

1. $a_t := a_0 + |F_E|/2$
2. $\dot{F}_E := F_E$
3. $\dot{F}_E[a_t] := F_E[a_t] \oplus 1$
4. **run** $\dot{F}_E$
5. **receive** m_t
6. **if not received then go to** 8
7. **if** $m_t \neq m$ **then stop**
8. $a_t := a_t + |m|$
9. **if** $a_t < a_0 + |F_E|$ **then go to** 2

Assume we have successfully received the message "*Secret firmvare v.1.0|n*" (note the '*v*' instead of '*w*'). This implies that our message originates at $a_m = a_t - 0xB$. Given that the plaintext m is known, we can extract a 17-byte segment of the keystream. Thus, we can write a properly encrypted null byte at a_m, followed by an encrypted code snippet – ideally one that dumps some memory location to UART.

Now, we need to transfer control to this code, which will require an extended series of experiments. If the message m is printed using the *printf()* function, we can replace it with an encrypted format string, "*%p%p...%p*", to obtain a stack dump. This dump will likely contain the saved contents of the link register *LR*. Consequently, *LR-4* corresponds to the address of a *BL* instruction. If this instruction directly calls a subroutine [5], it makes sense to conduct an iterative search for its encrypted representation using the mask 0x3FF07FF (*imm10:imm11*). Alternatively, if the address of the *Reset Handler* is known, we can attempt to iteratively modify it.

Certain optimization hacks can be applied. For instance, replacing the terminating '*|0*' in m allows decrypted memory content to be printed up to the next null byte. If successful, this trick may be repeated multiple times, hopefully without affecting firmware functionality. As a result, a sufficiently large memory block can be obtained for further experiments, along with a known portion of the keystream. We can place a sequence of *NOPs* at the beginning of this block, eliminating the need to modify the lower address bits. Once successful, we can then determine the exact XOR value by altering these lower address bits.

This is just one possible technique, but it demonstrates how firmware encryption based on AES-CTR could be compromised.

6. Conclusions

Recently introduced "flashless" microcontrollers, while offering the flexibility to choose the size of external flash memory, exhibit significant security vulnerability. A man-in-the-middle (MITM) attack can compromise the signature verification process, enabling

attackers to deploy patched firmware to achieve their objectives. While this study is purely theoretical at present, the authors intend to conduct experimental validation on real hardware in future work. This attack remains feasible even when the firmware is encrypted using AES-CTR or similar mechanisms. Although a hardware-based mitigation method has been proposed, the issue cannot be reliably addressed through software alone. Moreover, the MITM approach provides attackers with the capability to pinpoint more precise moments for fault injection attacks.

An interesting observation is that the exploit concept described is *not* applicable to the Cryptographic Embedded Controller CEC1712 from Microchip Technology Inc [11]. The CEC1712 secure bootloader authenticates and optionally decrypts the SPI Flash OEM boot image using AES-256, ECDSA P-384, and SHA-384 hardware cryptographic accelerators. Since XIP functionality is not implemented (code is decrypted to SRAM and executed from there), the described MITM attack is impossible, as is the decryption attack – if AES-CBC is used (which remains unspecified by the vendor).

References

1. Raspberry Pi Ltd. RP2040 datasheet: A microcontroller by Raspberry Pi / Raspberry Pi Ltd. – 2021.
2. NXP Semiconductors B.V. LPC18S50/S30/S10 32-bit ARM Cortex-M3 flashless MCU with security features. Product data sheet / NXP Semiconductors B.V. – 2020.
3. Infineon Technologies AG. CYW20829 AIROC™ Bluetooth® Low Energy 5.4 MCU / Infineon Technologies AG. – 2025.
4. Winbond Electronics Corporation. W25Q128JW 1.8V 128M-BIT Serial Flash Memory With Dual, Quad SPI / Winbond Electronics Corporation. – 2023.
5. Shcherbyna M. Improving Code Compression for ARM Cortex M Microcontrollers Using Pre-Filtering / M. Shcherbyna // Visnyk Nacional'nogo universitetu "L'vivs'ka politehnika". Ser. Information Systems and Networks. – 2023. – Vol. 14. – P. 225–234.
6. den Herrewegen J.V. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis / J.V. den Herrewegen, D. Oswald, F.D. Garcia, Q. Temeiza // IACR Transactions on Cryptographic Hardware and Embedded Systems. – 2020. – Vol. 2021, № 1. – P. 56– 81.
7. MCUboot. [Electronic resource] / MCUboot. – March 5, 2025. – Available at: https://github.com/mcu-tools/mcuboot
8. Maxwell B. Analysis of CRC methods and potential data integrity exploits / B. Maxwell, D.R. Thompson, G. Amerson, L. Johnson // International Conference on Emerging Technologies. – 2003.
9. Krovetz T. UMAC: Message Authentication Code using Universal Hashing / T. Krovetz // RFC 4418. – 2006.
10. Cysecuretools. [Electronic resource] / Cysecuretools. – March 5, 2025. – Available at: https://github.com/Infineon/cysecuretools/tree/master
11. Microchip Technology Inc. CEC1712 Data Sheet / Microchip Technology Inc. – 2020.

Article: received 11.09.2024
revised 16.10.2024
printing adoption 25.10.2024

УСУНЕННЯ ВРАЗЛИВОСТЕЙ БЕЗПЕЧНОГО ЗАВАНТАЖУВАЧА У МІКРОКОНТРОЛЕРАХ БЕЗ ВНУТРІШНЬОЇ ФЛЕШПАМ'ЯТІ

М. Щербина, П. Венгерський

Львівський національний університет імені Івана Франка,
вул. Університетська 1, Львів, 79000, Україна
e-mail: mykola.shcherbyna@lnu.edu.ua, petro.venherskyy@lnu.edu.ua

Розглянуто можливі атаки типу "людина посередині" (MITM) на безпечний початковий завантажувач у мікроконтролерах без внутрішньої флешпам'яті за допомогою пристрою, який контролює комунікацію шиною SPI. Продемонстровано можливість обходу криптографічної перевірки підпису вбудованого програмного забезпечення за допомогою точнішого визначення моменту ін'єкції помилок, а також запуск модифікованої прошивки безпосередньо після перевірки підпису. Проілюстровано обмежену можливість модифікації коду, зашифрованого AES-CTR, без знання ключа шифрування. Запропоновано апаратний захист вбудованого програмного забезпечення від описаної MITM-атаки під час виконання за допомогою кодів автентифікації.

Ключові слова: мікроконтролер, флешпам'ять, безпечний завантажувач, MITM, цифровий підпис, коди автентифікації повідомлень, AES-CTR, дешифрування.