ret2rev

Texas A&M University

Abhishek Bhattacharyya, Justin Block, Ryan Brasseaux, Cormac Cupples, Liam Haber, Danny Hernandez, Luke Loera, Emily Murphy, Nathan Nguyen, Mark Poveda, Bode Raymond, Lane Simmons, Anna Slater, Derek Viet, and Rohan Viswanathan

Advised by: Dr. Martin Carlisle, PhD

Design Overview

We utilized a two-stage bootloader, offloading critical functionality to SRAM to mitigate patching from the flash trojan. To communicate with the bootloader, each host-tool authenticated tself by signing a random challenge using ECDSA, with different signing keys for unprivileged and privileged functionality. Secure packages were created by accumulating package metadata, then signing and hashing each component. Components with confidentiality requirements were encrypted with XChaCha20Poly1305 to prevent IP disclosure and malicious tampering. This ensured all aspects of the CIA triad were met.

Defensive Highlight

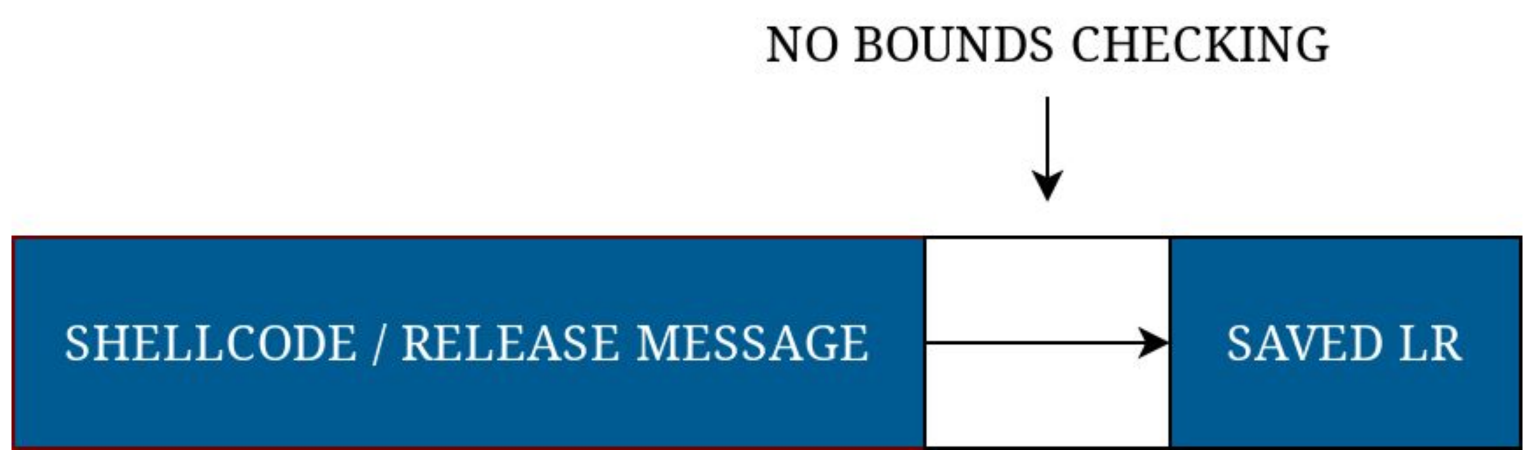
Our defensive highlight was our two-stage boot process. The first stage handled early peripheral initialization and decryption of the second stage in SRAM, which contained all critical logic. This ensured immutability of critical code during runtime. The second stage contained a decryption key to access the EEPROM keystore. After flash writes, the hash of the first stage was recomputed to ensure tampering had not occurred. In particular, if the flash trojan modified first stage bootloader code, the second stage would zeroize the second stage decryption key stored in EEPROM, which would brick the device.

This strategy was intended to mitigate the flash trojan and ensure all aspects of CIA triad were met. At-rest encryption used XChaCha20Poly1305, which has AEAD capabilities. This allowed us to ensure confidentiality and integrity of critical bootloader code. EEPROM was utilized as an encrypted keystore for signatures, hashes, nonces, and other metadata pertaining to installed images and device state.

Some flaws that may be present, but we have not tested, include:

- destructively overwriting the first stage and power cycling before the verification check
- binary patching critical return values in functions that we could not fit into SRAM (e.g. hashing and crypto functions)

Figure 1: Buffer Overflow



Offensive Highlight

The reference implementation had a buffer overflow in the uart_readline function, so attackers could write arbitrarily large release messages to overwrite the saved link register and gain control of the program counter (see Figure 1). Several designs did not patch this vulnerability. For our exploit, we injected shellcode in place of a true release message, then hijacked the PC to return to the beginning of the message in flash. By carefully crafting our shellcode to avoid instructions and immediates with nulls or newlines, we had arbitrary code execution. To actually obtain the flags, our payload leveraged existing functions within the binary to dump the entirety of EEPROM to SRAM, then wrote it to UART for further processing offline. Since all cryptographic keys were stored in EEPROM, it was relatively trivial to decrypt the provided packages and forge malicious ones.

To mitigate this attack, implementers should add bounds checking to the uart_readline function. This can be done by passing the size of the buffer into the function, then checking the loop counter against the buffer size on every iteration. By exiting the loop if the counter exceeds the buffer size (taking special care to account for the null terminator), attackers are prevented from overflowing the buffer and hijacking control flow.

Alternatively, implementers could patch this vulnerability by doing away with null-terminated strings entirely and modifying the protocol to pass the length of the release message upfront. This would have the added benefit of allowing the bootloader to enforce stricter validation on the size of the release message ahead of time.