Mass Ruby

University of Massachusetts Amherst

Ryan Lee, Nathan Costa, Alon Trogan, Vihar Vasavada, Gilbert Hoermann
Advised by: Professor Wayne Burleson

Design Overview

- Firmware and Configuration files stored and sent in roughly 1 KB encrypted chunks
- Bootloader implemented mostly in Rust while host tools were written in Python
- Bootloader computes its own hash and uses it as part of key derivation to detect Flash Trojan tampering
- Uses ChaCha20-Poly1305 AEAD[6] stream cipher instead of AES-based scheme. Readback implemented with HMAC-based challenge-response protocol.

Defensive Highlight

Defensive feature:

The design implemented a chunk system for the firmware and configuration files.

Goals of this feature:

By doing this, the design could load the files one chunk at a time, without having to store the entire thing in memory. This would enable the device to continue working, even if an attacker attempted to upload an invalid file.

Results:

While this approach seemed to work well, a fatal flaw that was discovered was chunk swapping. An attacker could swap the first chunk of either the firmware or configuration files with that of another file. While the design can prevent chunk reordering, it cannot prevent chunk swapping between files. Since the same key is used for firmware and configuration, swapping the first chunk of different firmwares (which would include their version numbers) would allow the attacker to obtain the rollback flag.

Improvements:

To fix the above design flaw, there are a number of solutions that we could have used: using a different key for each firmware/config, computing a hash over the entire firmware/config and verifying it (e.g. with a digital signature), and including some kind of tag with each chunk that ties it to the overall file. The first approach has its own issues, namely that many AEAD algorithms (including ChaCha20-Poly1305) do not commit to their keys and that we would need committing AE to fully guarantee its security. The second approach is something we considered but did not implement due to performance issues, though we would have tried harder to get it to work if we realized the importance of authenticating the entire file. The third approach would not have performance penalties, and it may be possible to use a mostly-shared nonce as the tag.

Offensive Highlight

Length Extension Attack:

In the attack phase, we tried attacking designs using a length extension attack on readback configurations.[1] The counter on one of the design's was not completely random and being initialized at 0. This counter was being concatenated as-is with the message string, rather than being hashed and concatenated. This made their encryption vulnerable to a length extension attack, as they were only using SHA256 instead of HMAC-SHA256, which is not vulnerable to this attack. Unfortunately, the attack was unsuccessful. The issue was that while we understood the concept of the attack, we couldn't implement it before the end of the competition.

Countermeasure:

A proposed fix for their code would be to hash the counter before concatenating it with the rest of the message. HMAC-SHA256 is not vulnerable to the same type of length extension attack, and thus would have made their design more secure.

References

1. https://github.com/marcelo140/length-extension
2. https://docs.rs/hmac/latest/hmac/
3. https://research.nccgroup.com/wp-content/uploads/2020/02/NCC-Group-Whitepaper-Microcontroller-Readback-Protection-1.pdf
4. https://docs.python.org/3/library/hmac.html
5. MAC and HMAC simply explained (with JavaScript snippets) | by Gonzalo Ruiz de Villa | gft-engineering | Medium
6. https://blog.cloudflare.com/it-takes-two-to-chacha-poly/