Wh1t3h4t5

Singapore Management University

Authors: Cheah King Yeh, Won Ying Keat
Advised by: Mr Lee Yeow Leong

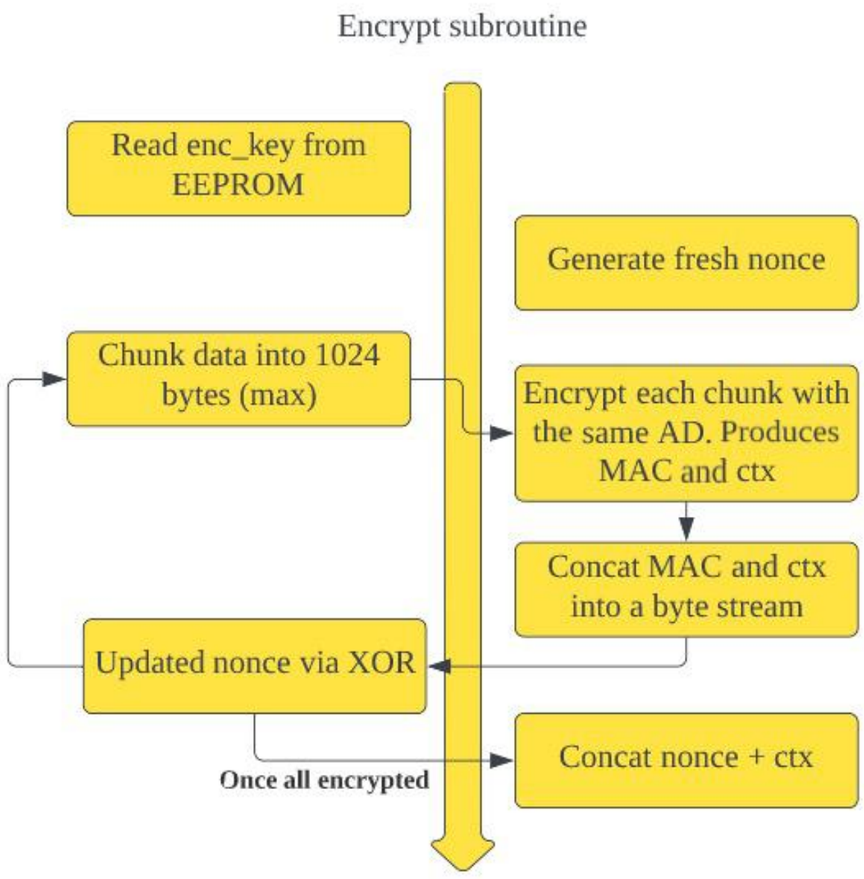
Design Overview

- Utilized XChaCha20-Poly1305 for authenticated encryption to ensure confidentiality and integrity of data
- Used Blake2b hashing to generate Message Authentication Codes to preserve data integrity
- Challenge-response scheme for authenticated access to protected resources to ensure availability
- Wiped and zeroed sensitive data whenever possible

Defensive Highlight

Chunked encryption and decryption

- XChaCha20-Poly1305 (RFC 8439[1])
- Allows for speedy bulk encryption of firmware and data
- Built in AEAD construction for metadata verification
- Implemented using *monocypher* – small, portable and fast – ideal for embedded systems[2]
- Nonce generation inspired by TLS1.3[3] using fixed IV XORed with sequence number
- Allows for maximum 256 chunk encryption with a nonce and key pair



Did it work?

- Yes, our design worked.
- However, our team's design could be improved:
  - Reconsider the need for chunked encryption

Considerations for the future

- Simplify overall encryption routine to exploit benefits provided by stream ciphers
- Seek inspiration from established real-world applications like TLS1.3 for better designs

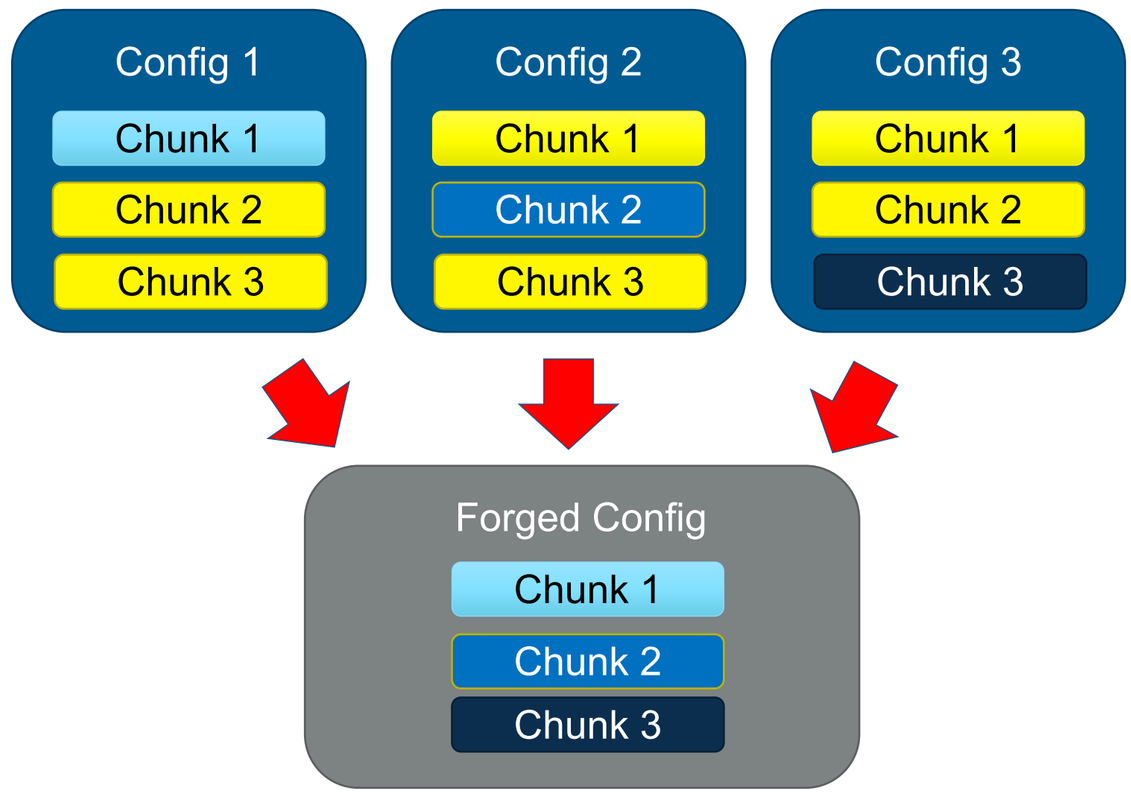
Offensive Highlight

Defeating chunked encryption and decryption

- Data is chunked, encrypted and verified individually when decrypted
- Each chunk consists of:
  - Associated Data (AD)
  - Nonce
  - Ciphertext
  - Tag for that chunk
- Attacking the implementation of its *associated data (AD)*
- AD contains:
  - Chunk number
  - Unused padding size
  - is_last_chunk (bool)
- We are provided with 3 different sample config files; each config contains 3 chunks based on team's design

Forging valid configs & exploiting checking mechanisms

- Manually concatenating chunks sampled from different valid config files
- Positioning chunks such that the associated data is valid
- Since tags are verified individually, such attacks will not raise an error
- Passes all identity checks and firmware happy uses it



Proposed Fix(es)

- An addition of a tag calculated based on the entire configuration file
- Bind & chain the chunks together, by including the previous chunk's tag into the next chunk's AD

References

1. https://datatracker.ietf.org/doc/html/rfc8439
2. https://monocypher.org/
3. https://datatracker.ietf.org/doc/html/rfc8446#section-5.3

Intro

Our team would like to present some of the key highlights that we managed to achieve throughout this competition. We would first walk through our overall design to provide an overview and intuition of the choice of the design. Next, we would discuss in detail one of the defensive mechanisms we utilized. Lastly, we would showcase an attack which we carried out on another team.

Design Overview

Firstly, for our team's design, we utilized state-of-the-art XChaCha20 with Poly 1305 encryption scheme for the Authenticated Encryption (AEAD). This is specially chosen to ensure confidentiality and integrity of the data, while allowing for an optional addition of associated data which we can transmit in plaintext while still verifying their integrity. Furthermore, to check for tampering post-decryption, blake2b hash is used to generate Message Authentication Codes (MAC) for verification. Lastly, to fulfill the requirements for availability of readback data, a challenge-response scheme is adopted for access by authenticated personnel. We also made sure to zero and wipe any sensitive data in the buffer whenever possible.

Defensive Highlight

Moving onto the defensive aspect, we wish to showcase our chunked encryption and decryption design which we adopted to protect the necessary data. Our basic research shows that XChaCha20-Poly1305 allows our team to attain many of the security requirements while having the convenience of built-in metadata (AD) verification. Moreover, there are popular and trusted libraries, in which we chose to use *monocypher* since it is small in size, portable and relatively fast when compared to other similar cryptographic implementations for this cipher. These factors make it ideal for embedded devices.

Arguably, any cipher is safe only if the nonce-key pair is never reused. Our team faced this issue since we chose to chunk our data and encrypt them individually. Instead of generating a new nonce for each chunk, we took inspiration from TLS1.3, where the nonce for each chunk is calculated based upon a fixed IV XORed with the sequence number, a fancy name for a running counter. In detail, the 10th byte of the IV is XORed with the sequence number to generate the nonce. Hence, this method means that we only need to transfer the nonce once and the same mechanism can be used to perform encryption and decryption properly. However, using only the 10th byte means that a maximum of 256 chunks can be encrypted using a single nonce-key pair. Despite the limitation, since the requirements only define a maximum data size equivalent to 64 chunks, we took the risk to stick with this design.

You can see from this diagram which showcases our team's encryption routine in detail. But the key question remains: Does it work?

We believe that there is still a lot of room for improvement. The first is to reconsider the actual need for data chunking before encryption. Since XChaCha20 is a stream cipher, perhaps encrypting the entire data in bulk would simplify the entire process and reduce attack vectors from possible poor chunking implementations. In addition, we could have sought inspiration from established real-world applications such as TLS1.3 to create a more robust overall design.

Offensive Highlight

Some say a good offense is also a good defense. Others say that if you know thyself and know thy enemy, you need not fear the results of a hundred battles. With that in mind, we decided to showcase how we attacked a similar implementation mentioned in our defensive portion - the chunked encryption design.

In this section, we managed to attack some implementation flaws, defeating the strong encryption and verification mechanisms put in place. As an introduction, the team's design allowed for chunks to be encrypted, decrypted and verified individually. Also, modern AEAD is used to provide confidentiality and data integrity. Within each chunk, it contains the associated data (AD), nonce, ciphertext as well as a verification tag acting as a MAC. Within the AD, it contains the chunk number, unused padding size as well as a boolean value indicating if it is the last chunk. We believe that the AD was put in place to ensure that the data cannot be reordered within a single configuration file, while providing critical metadata for the bootloader to take reference from.

Here, we attacked this specific AD implementation to exploit the trust that the bootloader has on the AD. Specifically, the bootloader will verify every chunk of data by decrypting the ciphertext and ensuring that the tag corresponds to the decrypted data with the provided AD. Using this knowledge and the 3 provided protected configuration files as samples, we can craft our attack to forge a "valid" configuration by exploiting the innate checking mechanisms.

We achieve this by manually crafting the configuration file through concatenating chunks sampled from the 3 different provided valid configs. Since each individual chunk remains unedited, the bootloader will not return an error when decrypting and validating individual chunks. Hence, we can position the chunks such that the chunk number and is_last_chunk metadata is correct. We have provided a simple visualization to showcase how it is done. In this example, the forged config samples chunk 1, 2 and 3 from config files 1, 2 and 3 respectively. This allows us to craft a fake config which passes all checks, eventually getting loaded into the firmware and causing integrity issues, granting us the "abort" flag.

Although this issue is severe, there is a simple fix. The team can consider adding MAC verification by generating a tag from the entirety of the config file instead of using individual chunks. Another method to consider is an approach similar to blockchain, in which collision-resistant information from the previous chunk (such as the calculated tag) can be included into the AD of the next chunk, preventing an adversary from collecting and reassembling the chunks from different valid config to craft a fake config.

Conclusion

With this, we have come to the end of our showcase and we hope you enjoyed our walkthrough.