TheMuffinMob

Worcester Polytechnic Institute

A. Ames, J. Backer, K. Jesse, K. Kaufman, H. Kyriacou, I. Robinson

Advised by: Prof. Robert J. Walls

April 24, 2023

Design Overview

All board-to-board traffic was encrypted and authenticated using **XChaCha20-Poly1305**, using symmetric keys unique to each car/fob pair (in order to ensure that only the right fobs could communicate with a car - see **SR1** [1] + **SR4**) as well as random nonces (to prevent* replay attacks - see **SR2 + SR3**), as shown in figure 1.

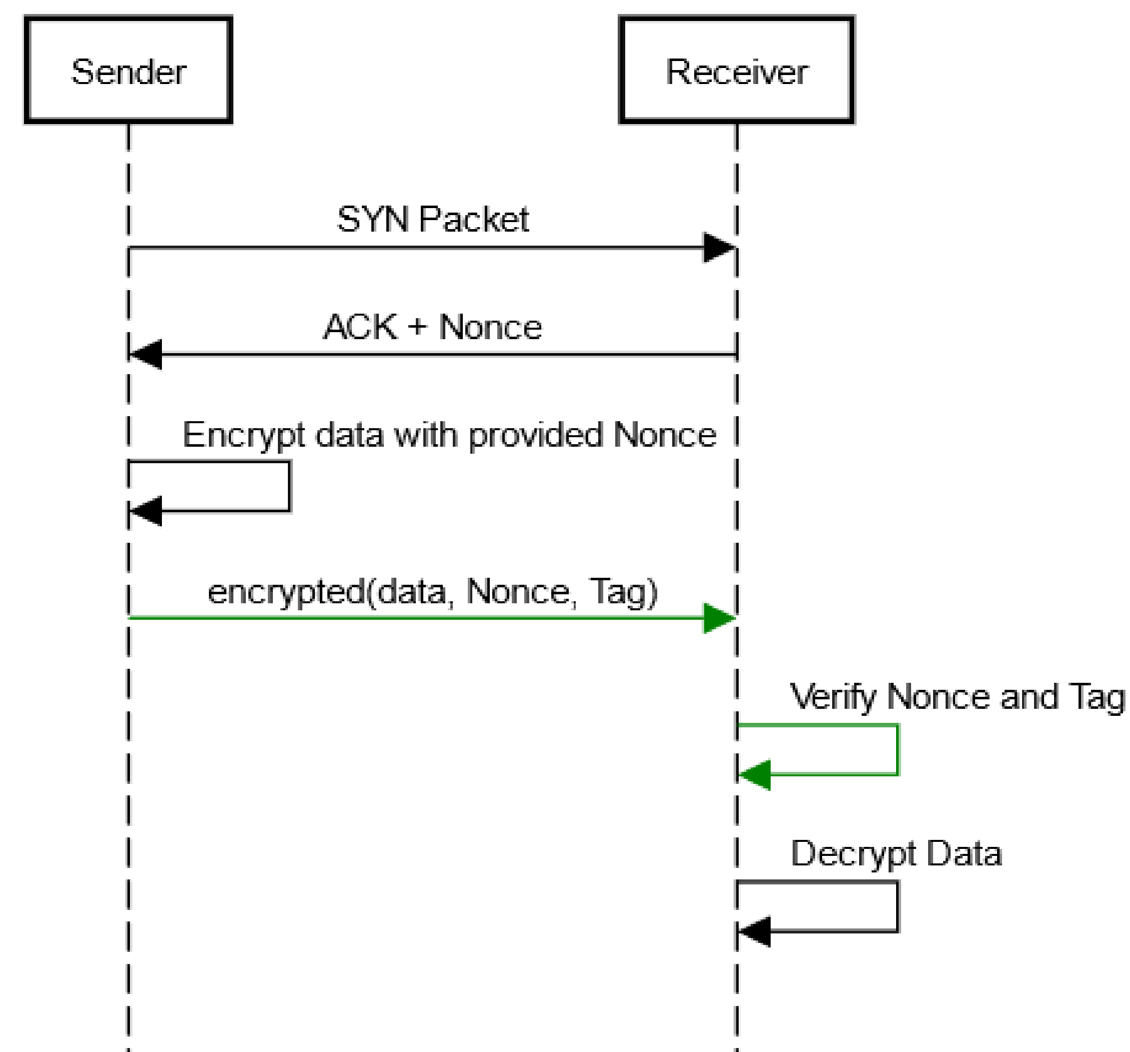
During pairing, an unpaired fob and paired fob would perform a key exchange in order to generate a symmetric key for future use. This ensured forward secrecy, keeping their communications secure against sniffing attacks.

Packaged features were signed with a manufacturer-held, car-specific private key to ensure authenticity (see **SR5 + SR6.**)

Some secrets (such as pairing PINs) were treated with special care, being fed into **key derivation functions** in order to generate keys that could be used to encrypt "success messages" at build time and decrypt/verify those messages at runtime. Since the original secrets would be destroyed or otherwise inaccessible to the attacker, our design was extremely secure against secret-extraction attacks.

* ***This was not achieved***, because our RNG did not incorporate environmental noise. As a result, it would produce the same output sequence after the firmware was reinstalled. We intend to improve upon this for our future designs.

Figure 1: Encrypted protocol



Defensive Highlight

A defensive feature that truly set our design apart from almost all of the others (besides our use of Rust) was our protection against fault injection attacks. While we recognized the difficulty of executing a fault injection attack in practice, we felt it was prudent to at least consider the possibility of one occurring. A successful fault injection attack – however unlikely - could be disastrous, potentially compromising the system and breaking one (or more) of its security goals.

To mitigate these attacks, we made use of "double-down if-statements", or "double-downs" for short (see: **embed/src/security/anti_glitching.rs**.) Double-downs evaluate the condition *multiple times* in order to detect any unusual behavior. If a fault is detected, the entire system aborts and requires a reset. This is a low-overhead method of managing fault injection attacks that has some scientific backing [2], although it is not a complete solution.

Double-downs worked very well for us, but in the future, we may seek to introduce supplementary countermeasures, such as random delays to thwart attacks that rely on precise timing.

Offensive Highlight

Before the attack phase even began, we identified a vulnerable function in the reference design that we suspected many teams would use without any additional review: **uart_readline** [3], which is practically equivalent to the infamous **gets** [4] function from the C standard library that allows for buffer overflows.

The typical attack flow was as follows:
1. Identify the path of execution that leads to a call to **uart_readline**. For several designs, this was found in the fob firmware – usually in the pairing or feature-enabling code.
2. Load an unprotected target firmware image onto an unkeyed board, and work within a debugger to identify the number of bytes needed to overwrite the saved return address. In the process, we would also identify any additional addresses we needed, such as the location of the buffer containing our input.
3. Develop an exploit payload to dump data (firmware or EEPROM contents) over the host UART connection. A firmware dump was often sufficient to obtain all the information we needed (such as fob pairing PINs.) Firmware dumping could be achieved using a single call to **uart_write** (also from the insecure example), since the firmware is mapped into memory.

This attack was successful (albeit more difficult) even against designs that leveraged the Memory Protection Unit (MPU) to prevent stack shellcode execution. Those designs could be defeated using return-oriented-programming, or ROP chains, in which *existing* code was bent to our will.

This attack could be prevented by modifying **uart_readline** to perform bounds checking, à la **fgets** [5] in C. As it turns out, some teams did just that, forcing us to find other attack vectors.

References

1. Challenge Design Summary v1.1, pp. 16
2. Spensky et al., "Glitching Demystified"
3. uart.c line 104
4. C library function - gets()
5. C library function - fgets()