ESC 2019 Final Report

Adam Verner
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
vernead2@fit.cvut.cz

Jan Havránek
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
havraja6@fit.cvut.cz

Eliška Kühnerová
Faculty of Informatics and Statistics
University of Economics
Prague, Czech Republic
kuhe00@vse.cz

***Abstract*—This document presents the best strategies, methods and procedures used for solving challenges presented to finalists of ESC 2019[1].**

I. Introduction

In the first part of this document we described the architecture of the hardware RFID board that we received as part of the Embedded Security Challenge. Next, we discussed our approach to reverse engineering and techniques that were common for most or all challenges that we were presented with. Lastly, we described our solutions for every solved challenge. Each challenge is broken down in an individual section.

II. System overview

The board schematic was at first reversed by hand using microscope and ohmmeter, but later on the official schematic was released, see appendix A.

The board consists of two microcontrollers (one on development board), RFID card reader, OLED display four buttons and 2 LEDs. There are two communication buses on the board, I2C and SPI. When flashing an empty board, the Teensy is first programmed as an ISP(in system programmer) which is then used to load the memory to Atmega chip, then the Teensy is programmed with it's intended firmware. The SPI is not only used for programming the Atmega, but also by the Atmega to communicate with the card reader.

Cumulatively there are 18 individual challenges with various difficulty ranging from very easy to really hard.

When running the Atmega is the master device, who controlls everything. When user selects "start" it waits for RFID card. after reading 1024 bytes from the card the Atmega sends the data, challenge number and buttons to the Teensy which then starts selected challenge with corresponding data. The structure of the received packet could be described as C structure:

```
struct packet {
    char comm;
    char RFID[1024];
    char keys[48];
    char buttons;
    char challengeNum;
};
```

Fig. 1. single packet structure

The variable *buttons* represents both inputs A and B as it's nibbles, such as A is the higher nibble and B is the lower part.

A. Teensy firmware

The firmware for the Teensy board is an 32-bit ARM with debug info compiled in. The debug info is handy as it makes the output from decompilation step much nicer. For example, when selecting *Program Trees -¿ DWARF* in Ghidra it displays all files the binary was compiled with. Most interesting one here is *TeensyChallengeSetX.ino*. Not only is it's presence telling, that it was compiled using Arduino IDE[2], it also contains function called *challenge_x*.

B. Atmega firmware

Atmega handles the communication with the peripherals, the user and verifies output from the selected challenge. Apart from that, there does not happen anything much interesting at all, except for one challenge (blue, described in section XI). All solutions, except one are solvable without even opening the binary for Atmega.

III. Approach and techniques

A. Ghidra

Main tool we used for reverse engineering the individual challenges is Ghidra, an open source SRE tool from the NSA. It offers very powerful set of reversing tools along with a quite solid decompiler for various architectures. The most important aspect for us in the context of ESC 2019 is that Ghidra can decompile ARM binaries.

Very important step that we performed before reverse engineering each challenge was to correctly set up the function parameter, the RFID packet structure, so that Ghidra's decompiler correctly recognized when data was read from or written to the packet.

Code samples in this document are mostly parts of reverse engineered challenge functions using the Ghidra decompiler with some changes by hand.

[1] https://github.com/TrustworthyComputing/csaw_esc_2019

[2] https://en.wikipedia.org/wiki/Arduino_IDE

B. Radare2

Another very powerful tool that we used while solving the challenges was Radare2. Similar to Ghidra, it is a very powerful tool for reverse engineering various binary executable formats, etc. Although it by default does not have a decompiler, it is very versatile in terms of examining and patching binary data and assembly instructions. This is the reason we used it to patch certain binaries mostly to print and examine internal state of some of the challenges where static reverse engineering was not enough.

IV. Challenge 0 Lounge

The first challenge in the set. There is a lot of things happening in the challenge function, mainly mathematical operations with 64 bit floating point numbers. Luckily, most of the code can be ignored as only a few lines of code are important for the solution. The final comparison which determines if the challenge was solved correctly is:

$$x * y == 6319$$

The number 6319 has only two factors, 71 and 89, which are the values that need to be saved in x and y in order to successfully solve the challenge. Therefore, we need to backtrace how the variables are set. Figure 2 shows the important code.

```
doubleX = __aeabi_i2d(
    (packet.RFID[76] >> 4 & 1) * 1+
    (packet.RFID[77] >> 6 & 1) * 2 +
    (packet.RFID[77] >> 3 & 1) * 4 +
    (packet.RFID[77] >> 0 & 1) * 8 +
    (packet.RFID[76] >> 7 & 1) * 0x10 +
    (packet.RFID[76] >> 1 & 1) * 0x20 +
    (packet.RFID[77] >> 7 & 1) * 0x40);

doubleY = __aeabi_i2d(
    (packet.RFID[77] >> 5 & 1) * 1 +
    (packet.RFID[76] >> 0 & 1) * 2 +
    (packet.RFID[77] >> 1 & 1) * 4 +
    (packet.RFID[76] >> 6 & 1) * 8 +
    (packet.RFID[76] >> 5 & 1) * 0x10 +
    (packet.RFID[76] >> 2 & 1) * 0x20 +
    (packet.RFID[77] >> 5 & 1) * 0x40);

int x = __aeabi_d2iz(doubleX);
int y = __aeabi_d2iz(doubleY);
```

Fig. 2. Code which sets values for variables x and y

Per the ARM run-time ABI documentation, the function *__aeabi_i2d* converts an integer value to double and function *__aeabi_d2iz* converts double to integer using C-style conversion. Basically it just returns the integer part of the floating point number.

The bits of 76th and 77th byte of the card memory have to be set in such a way that the bit shifting operations shown in figure 2 evaluates to the desired values for x and y. We can achieve that by reversing the bit shifts, as shown in figure 3.

```
x = 71
y = 89

rfid76 = (((x >> 4) & 1) << 7) |
         (((y >> 3) & 1) << 6) |
         (((y >> 4) & 1) << 5) |
         (((x >> 0) & 1) << 4) |
         (((y >> 5) & 1) << 2) |
         (((x >> 5) & 1) << 1) |
         (((y >> 1) & 1) << 0)

rfid77 = (((x >> 6) & 1) << 7) |
         (((x >> 1) & 1) << 6) |
         (((y >> 0) & 1) << 5) |
         (((y >> 6) & 1) << 5) |
         (((x >> 2) & 1) << 3) |
         (((y >> 2) & 1) << 1) |
         (((x >> 3) & 1) << 0)
```

Fig. 3. Code which calculates correct RFID bytes for lounge challenge

This challenge actually has two possible solutions as the values of variables x and y can be swapped and the product is still the same.

V. Challenge 1 Closet

First thing this challenge does, is that it saves the string "ESC19-rocks!" on stack to location R7+0x40 where, similarly to other challenges, R7 is used as a base register for variables.

Next, as shown in figure 4, bytes received from the RFID card on indexes 92 to 115 are copied to an array on stack to location R7+0x28. Values of the loop variable (here named i) are printed to the serial port.

```
uint8_t data[24];        // R7+0x28
for (int i = 0; i < 24; i++) {
    data[i] = RFID[i+92];
    Print::println(Serial, i);
}
```

Fig. 4. Code which copies data from card memory to stack

Lastly, the bytes from RFID card are used as offsets in a verification loop, shown in figure 5.

Therefore, our goal is to make the expression (R7 + 0x28 + data[i]) evaluate in such a way that the resulting pointer points to the initial string "ESC19-rocks!". We can do that by setting the first 12 bytes of *data* array to values [24, 25, 26, ..., 34, 35]. The result then points to R7+0x28+0x18 = R7+0x40 which is the same location as the desired char array and the verification loop is successful.

```
bool correct = true;
for (int i = 0; i < 12; i++) {
    if (*(R7 + 0x28 + data[i]) != str[i]) {
        correct = false;
    }
}
```

Fig. 5. Closet challenge verification loop

VI. Challenge 2 Cafe

Challenge cafe begins by definining two character arrays: mangled challenge success message and some other text, shown in figure 6.

```
char winText[] = "\xcb\x83\xc4\xa6\x93r"
                 "challenge\xb0\xe5"
                 "Z\xc7*4bcdefghi ";
char haxorz[] = "h4x0R2 dr00L";
```

Fig. 6. Challenge cafe initialization

We know that we need to demangle the message (here named *winText*) so that it says "solved challenge cafe abcdefghi ". The message has 12 bytes changed in total, 6 bytes at offset 0 and 6 bytes at offset 17.

```
for (int i = 0; i < 6; i++) {
    winText[i] ^= haxorz[i];

    if (((packet.RFID[90] >> i) & 1) == 0) {
        winText[i] -= packet.RFID[i + 78];
    } else {
        winText[i] += packet.RFID[i + 78];
    }
}
```

Fig. 7. Challenge cafe first code loop

Figure 7 shows the first loop that the challenge code contains. This loop changes the first group of mangled bytes. For each byte, it xors it with a character from *haxorz* array and then depending on bit i from p.RFID[90] adds or subtracts byte p.RFID[i + 78].

The second loop, shown in figure 8 does the same thing with minor differences in offsets. It operates with the 6 bytes at offset 17, select addition or subtraction by bits in byte packet.RFID[91] and adds or subtracts value from byte packet.RFID[i + 67].

We can solve this challenge by simply xoring the mangled message bytes with corresponding characters in the *haxorz* array and then calculating the difference between the resulting value and correct success message bytes. The code for determining the first 6 correct RFID bytes is shown in figure 9. Python code for the other 6 bytes is the same with different offsets, therefore it was omitted.

```
for (int j = 17; j < 23; j++) {
    winText[j] ^= haxorz[j - 11];

    if (((p.RFID[91] >> (j-17)) & 1) == 0)
        winText[j] -= p.RFID[j + 67];
    else
        winText[j] += p.RFID[j + 67];
}
```

Fig. 8. Challenge cafe second code loop

```
for i in range(6):
    win_text[i] ^= haxorz[i]

    diff = win_text[i] - win_text_right[i]

    RFID[i + 78] = abs(diff)
    if diff < 0:
        RFID[90] |= (1 << i)
```

Fig. 9. Solution code for first 6 bytes

VII. Challenge 3 Stairs

Challenge 3 solution walkthrough was made available by organizers[3]. That gives some insight in how the challenges are structured and what does the source code looks like. It also verified that all the solution logic is inside the Teensy binary

VIII. Challenge 4 Mobile

Challenge mobile starts with defining a character array which contains a few spaces and lowercase letters, then it initializes the classic success string with the catch that each character of the word "challenge" is replaced with spaces. Both are shown in figure 10.

```
char chars[] = "   abcdefghijklmnopqrstu";
char s[] = "solved           mobile abcdefg";
```

Fig. 10. Char array definitions for challenge mobile

The challenge then takes 16 bytes from card at offset 132 and extracts nibbles (4 bits) from the bytes. Then, depending on values of two consecutive nibbles, it performs an operation. The operations are listed below.

- increment index variable to *chars* array (j) if the nibbles are equal
- reset index variable j to 0 if one of the nibbles is 0

[3]https://bit.ly/33eFAbQ

- copy character from *chars* array at index $j + first\_nibble * 3$ to char array s at index k, increment k

Therefore we need to find such a sequence of bytes that the process described above copies the correct characters from *chars* array to *s* array so that the resulting string is "solved challenge mobile abcdefg". Finding of the correct byte values is performed by the script in figure 11.

```
al = '   abcdefghijklmnopqrstu'
assert len(al) == 24
nibbles = []
for c in 'challenge':
    i = al.index(c)
    j = i%3
    n = i//3
    nibbles += [n]*(j+1) + [0]

res = bytearray()
for i in range(0, len(nibbles), 2):
    b = (nibbles[i] << 4)
    b |= nibbles[i+1]
    res.append(b)

print(list(res))
```

Fig. 11. Challenge mobile solution script

IX. Challenge 5 Dance

This challenge takes 8 bytes from RFID at offset 0x83 and calculates their SHA256 hash. The calculated hash is then compared with hardcoded string.

There are many ways to discover the contents of the string. The easiest option would be using online resources e.g.: crackstation[4]. The second option is running local dictionary attack, which depends on what wordlist we use. The last option would be to run bruteforce attack.

The first option gives the string *password*. That's no surprise. The second method would be applicable if the hash was salted.

X. Challenge 6 Code

This challenge is similar to the previous one. The function takes one byte from RFID at offset 155 and appends it to constant string *imjustrandomdatathathasnomeaningwhatsoever!*. The output is then hashed using H45H function and checked against static string. Using hashanalyzer[5] the string is identified as MD5 hash. Reverse lookup yields no results. Writing a simple script to try every alphanumeric combination gives the result almost immediately: *imjustrandomdatathathasnomeaningwhatsoever!L*

[4]https://crackstation.net/

[5]urlwww.tunnelsup.com/hash-analyzer/

XI. Challenge 7 blue

Challenge blue, as the only one in the competition, required us to reverse engineer both the challenge function in Teensy firmware and the AVR firmware.

Firstly, the Teensy firmware part. In the beginning of the challenge function, there is a base64 encoded hint telling us that the blue tag we received should be used in this challenge. Reverse engineering the rest of the challenge function was quite easy. All it does, is that it takes 8 bytes from the card's memory at offset 960, hashes it using SHA256 and compares the result with a hash hardcoded in the binary.

Minding the hint, we focused on the blue tag.

Firstly, we tried to directly scan the tag which did not solve the challenge. Next, we examined the memory of the tag and discovered that it was preprogrammed with some data at the mentioned offset 960. Hashing the first 8 bytes of the data produced the correct hash. All we had to do after this finding was to make the RFID board read the data from the tag.

As we knew that the chip responsible for the tag scanning is the AVR, we reverse engineered its firmware and found that for this specific challenge, the AVR is authenticating with a Mifare key that is different from the factory default (= `FFFFFFFFFFFF`, used in other challenges). The key used for reading the tag's data is `5d6a0064f591`. Therefore, to solve this challenge, we set the access key A in block 0x3F to the aforementioned value using the *nfc-mfclassic* command line tool from libnfc library[6].

As it is required to change the card's sector access keys, this challenge cannot be solved with the provided sender.py script because it is unable to perform this operation.

XII. Challenge 8 Uno

This one of the hardest challenges in the competition. There is a simple hint *is OISC 1337* referencing the One Instruction set PC[7]. This knowledge will come in handy later.

After reversing the decompiled code from Ghidra it becomes clear that the challenge is modified version of Subleq[8] with some additional features.

- if $*b = -1$ one byte from $*a$ is appended to challHash
- if $ptr > 60$ or $*a = -1$ execution ends
- The memory is limited to 60 bytes
- modified answer string is appended to memory

In order to solve the challenge the *answer* string must be first modified and then written to *challHash*.

Pseudocode of what the Subleq code should do:

Writing a Virtual Machine and debugger to develop, debug and test the code locally, we get the final solution.

XIII. Challenge 9 game

This challenge requires the hacker to win or at least tie the already started game of *TIC-TAC-TOE*.

Ghidra sometimes has an issue with stack operations, see figure 13.

[6]https://github.com/nfc-tools/libnfc

[7]https://en.wikipedia.org/wiki/One_instruction_set_computer

[8]https://esolangs.org/wiki/Subleq

```
for x in {63..94}
    m[x] -= 3;
    challHash_write(m[x])
exit()
```

Fig. 12. script logic pseudo-code

```
v1 = evaluate((char (*) [3])board);
if (v1 / 10 == 0) {
  v2 = (uint)(r.RFID[i + 0x8c] >> 4);
  v3 = (uint)r.RFID[i + 0x8c] & 0xf;
  v4 = v3 + v2 * 3 + -0x48;
  if ((&stack0xfffffe0)[v4] == '_') {
      (&stack0xfffffe0)[v4] = 0x6f;
  }
```

Fig. 13. Decompiled board logic

After inspecting the stack, it's clear that the $stack0xffe0-0x48$ is equal to *board* variable which holds the 3x3 array with board plan.

Three moves were already played in such way that it's not possible to win, see figure 14, unless opponent makes a mistake. fortunately it's enough to tie the game.

Playing [2:0][9], opponent has to respond with [0:2]. Playing [0:1] to stop him and now he has to respond with [2:1]. The final move could be played either on [1:2] or [2:2] tying the game.

XIV. Challenge 10 Break

The first part of the challenge function is a loop with constant input and constant loop count, it will always yield the same output. In this case it is 0x4161

[9]coordinates in format [X:Y]

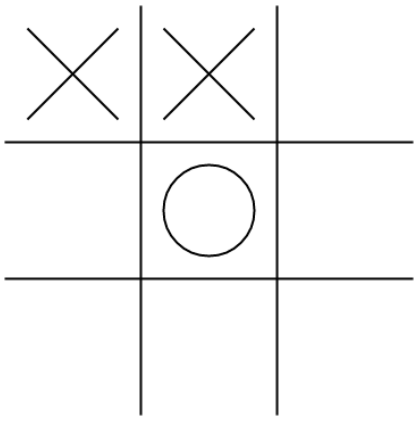


Fig. 14. Gameboard start

The second part of the challenge checks the output from the previous loop against computed value. there is an overlap with buttons state, but by setting them to 0, we can ignore them[10].

$$(r.RFID[159] \ll 8) \oplus r.RFID[160] \iff \text{0x4161}$$

Right away it's obvious that

$$r.RFID[159] = \text{0x41}$$

$$r.RFID[160] = \text{0x61}$$

XV. Challenge 11 Recess

Challenge 11 was one of the easier challenge in the set. Figure 15 shows the main code of the challenge. It simply takes 4 bytes of data from the RFID card from offset 161 and performs a CRC32 checksum on it.

```
for (int i = 0; i < 4; i++) {
    data[i] = packet.RFID[i + 161]
}
int crc = rc_crc32(0, data, 4)
```

Fig. 15. Code of challenge recess

Later in the challenge function the CRC value is checked with value 0x36476684 to determine if the challenge was solved correctly. Thanks to the fact that the CRC is calculated from only 4 bytes, it was feasible to use brute force and go through all possible combinations of the 4 bytes. By assuming that the 4 bytes will only be printable characters, we could significantly reduce the number of possibilities to check. And sure enough, the required 4 bytes of data say `g00d`.

XVI. Challenge 12 Steel

Ghidra reveals that this function takes three bytes, calculates one byte from them

```
v1 = RFID[401] - RFID[403] * RFID[402];
char center = v1 % 25 + 45;
```

This value then gets hashed 10 times, each round updating the hash from previous digest. The final hash then gets checked with hardcoded string From the implementation it's obvious that's it's MD5, but this implementation differs a little from other standard implementations[11], specifically, when using same instance after calculating the digest, the output starts to differ.

After patching the binary using Radare2[12], the Teensy loops through all 255 possible *center* combinations.

[10]$x \oplus 0 \iff x$

[11]e.g.: Python's*hashlib*

[12]rada.re

```
...
4765fae13f940050ded3e959089093e7  3e
18c41a53c2be6229deb7711f9d95c8c9  3d
1bb86483cd6dec71551793f94d1ac95f  3c
703224f765d313ee4ed0fadcf9d63a5e  3b <-
9329af69fd45bf4fb15abf5cbe3d1625  3a
d561e48fc0b225b1ed37a6120ab0efd3  39
7a44262a1ea0504030bebe8a674779a8  38
...
```

This challenge has more than 11 thousand possible solutions due to the nature of calculation done in step 1. The most simple one is

$$r.RFID[401] = 14;$$

XVII. Challenge 13 Caeser

To complete this challenge you must find x and k for specified Polynomials P_{x_1} and P_{x_2} such that $P_{x_1} = 0\text{x}428$ and $P_{x_2} = 0\text{x}1af6$ after calculating the system of equations, the results will be as follows: $k = 0\text{xaaaa}$ and $y = 0\text{xa9f2}$

Calculating the value of the third polynomial leads to value of $0\text{x}6142$

XVIII. Challenge 14 Spiral

Figure 16 shows the main challenge code that was imporant to reverse engineer.

```
uVar1 = uVar2 + byteswap(uVar1) ^ 0xbeef;
uVar2 = uVar1 ^ rol(uVar2, 3);
uVar1 = uVar2 + byteswap(uVar1) ^ 0x9bb0;
uVar2 = uVar1 ^ rol(uVar2, 3);
uVar1 = uVar2 + byteswap(uVar1) ^ 0xb499;
uVar2 = uVar1 ^ rol(uVar2, 3);

if (uVar1 == 0xa29f && uVar2 == 0xd481) {
    correct = true;
}
```

Fig. 16. Code of challenge spiral

Variables *uVar1* and *uVar2* are both 16 bits in size, *byteswap* denotes a function which swaps the two bytes of the variable, *rol* is bit rotate left and *ror* bit rotate right.

The variables are saved in bytes from 397 to 401 on the RFID card. Therefore, to solve this challenge, we need to set the mentioned bytes so that the variables evaluate to correct values after the operations above (0xa29f and 0xd481).

Thanks to the fact, that no information is lost during the operations, we can simply reverse their order to get the original values, like shown in figure 17.

The resulting values are 0xcafe and 0xfade which need to be save to memory of the RFID card to offset 397 as mentioned before.

```
uint16_t uVar1 = 0xa29f;
uint16_t uVar2 = 0xd481;

uVar2 = ror(uVar2 ^ uVar1, 3);
uVar1 = byteswap((uVar1 ^ 0xb499) - uVar2);
uVar2 = ror(uVar2 ^ uVar1, 3);
uVar1 = byteswap((uVar1 ^ 0x9bb0) - uVar2);
uVar2 = ror(uVar2 ^ uVar1, 3);
uVar1 = byteswap((uVar1 ^ 0xbeef) - uVar2);

printf("%hx %hx\n", uVar1, uVar2);
```

Fig. 17. Solution code for challenge spiral

```
Stack [-0x58] buffer
Stack [-0x54] max
Stack [-0x50] check
Stack [-0x4c] val
Stack [-0x48] *ptr
Stack [-0x44] buff2
```

Fig. 18. Spire Stack

XIX. Challenge 17 Spire

The Stack of the function can be seen if Fig. 18

There are two loops. The first loop writes eight bytes to 4byte array *buffer*, causing an buffer overflow and rewriting the value of *int max* and check. In order to solve the challenge the value of *check* needs to be a positive integer.

The second loop goes from zero to max decrementing the value at the end of each cycle. The loop writes whatever is in the RFID card to $buff2[loop\_counter]$ causing an buffer underflow. The underflow is controlled by value of *max* that was written in the previous loop.

The overwritten **ptr* needs to point to a valid address, because after the loop ends the challenge will try to write contents of d into whatever is at the address. If the address is invalid, the processor will halt itself. Valid addresses can be found in memory map table in the processors datasheet[13].

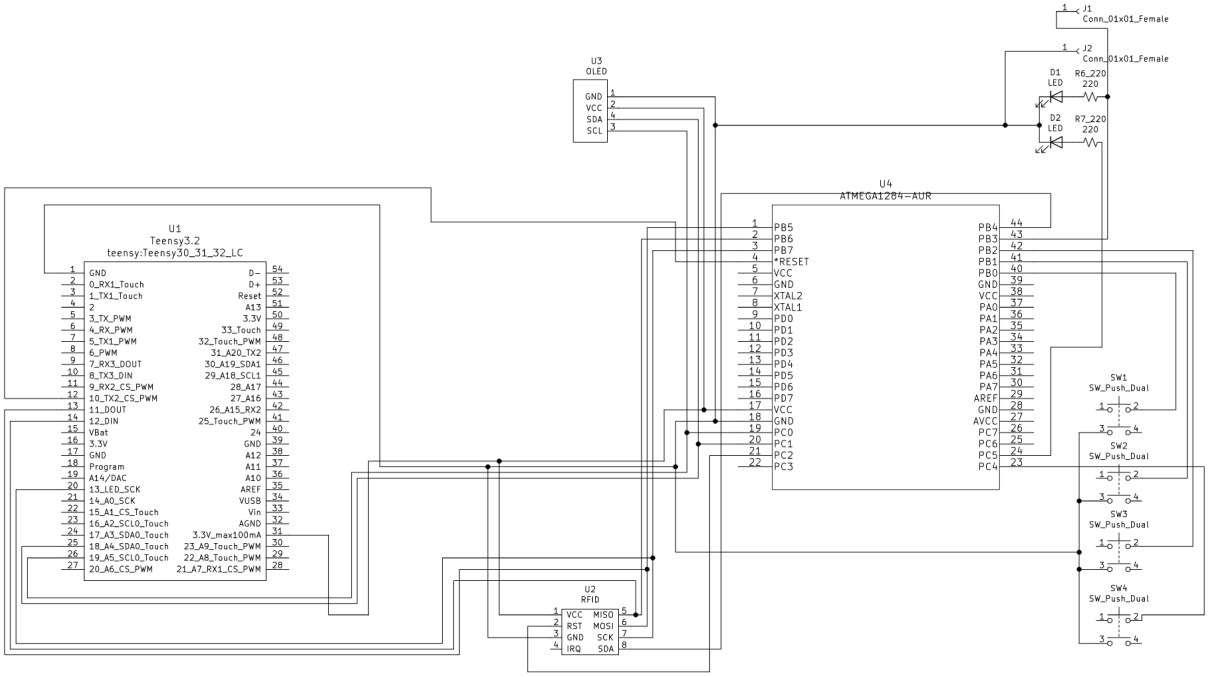
XX. Conclusion

In this paper we explained the architecture of the challenge board and the communication protocols it uses. We showcased various reverse engineering techniques with examples on real challenges and explained ways to save time and increase productivity using open source tools like Ghidra and Radare2. There is also breakdown for every solved challenge (16 out of 18) and their possible solutions.

[13]https://www.pjrc.com/teensy/K20P64M72SF1RM.pdf

Set	index	name	points	
A	0	lounge	100	‘643a6fa20b171fdf3a9e7e1975ce62892fde9cecf2056a73d85fa2d0802d3000‘
A	1	closet	100	‘293f7b60b994512db99836ae7d5bab88b2d0089f90fcf6d51b95b374200dc20f‘
A	2	cafe	130	‘d05235d380e913b5625d653c555de8925f249838896651a95bb35ea4e7863a5e‘
A	3	stairs	50	‘396f4b1cdf1cc2e7680f2a8716a18c887cd489e12232e75b6810e9d5e91426c7‘
B	4	mobile	100	‘f2f3792453040e837e7e1584e72859bfaa6b8c09d73d185be53b35886b6455c2‘
B	5	dance	130	‘e631b32e3e493c51e5c2b22d1486d401c76ac83e3910566924bcc51b2157c837‘
B	6	code	50	‘372ded6746e45ef7c8ad5a22c5738a4b5aa982da66bc8a426aa1cca830d05af3‘
B	7	blue		‘senderdoesnotworkwiththischallenge:(‘
C	8	uno	200	‘2e120f2237c71f18d29451c4787ac1df8285909618e2a821ee7c97d7efde246c‘
C	9	game	150	‘a536829856d84ccd53ff8bcf534a65c5678bdbe9ce20f78407e1c987ba517e8a‘
C	10	break	70	‘2d3448f09329f453e6f3a5403d89c061a9dabfbb9103ad6b8cc86d16345a7547‘
C	11	recess	100	‘370815b8d8fde829f5c35f893d0b4139d61a775baa4181fcac1fffe014bde9ea‘
D	8	bounce		
E	12	steel	100	‘5921d2ca353338c5f04c92205dc8f8bc8734f092a9e63e5f02ec106f7a7d99b4‘
E	13	caeser	150	‘551b5cff372d310b57d39b616400461be0a1450c519a2a542f33a7af0dd565f3‘
E	14	spiral	130	‘26ee8470c732dfc821bbe0561b446dc8086560e4e222b22e6a74e559d90a7d61‘
E	15	tower		
F	17	spire	150	‘f322344822257bb66542629db08bcea285411068963e30f0595847c79ef76f37‘

APPENDIX A



APPENDIX B