

# Team TRX: CSAW Embedded Security Challenge Final Report

Matteo Almanza  
*Computer Science Department*  
*Sapienza University*  
Rome, Italy  
almanza@di.uniroma1.it

Cristian Assaiante  
*Computer Engineering Department*  
*Sapienza University*  
Rome, Italy  
cristianassaiante@outlook.com

Qian Matteo, Chen  
*Computer Science Department*  
*Sapienza University*  
Rome, Italy  
chen.1756501@studenti.uniroma1.it

Christian Cotignola  
*Computer Engineering Department*  
*Sapienza University*  
Rome, Italy  
christiancotignola@gmail.com

Camil Demetrescu  
*Computer Engineering Department*  
*Sapienza University*  
Rome, Italy  
demetres@diag.uniroma1.it

**Abstract**—Firmware reverse engineering is a daunting task. Real world scenarios often provide IoT devices and firmware binaries without source code and understanding the inner functionalities is usually the main task to assess the security of software running on such devices.

In this document we will present the reader with the approach our team followed in order to reverse engineer, efficiently and precisely, all the ARM and AVR challenge binaries provided in the Embedded Security Challenge. Our approach is general enough to be extended to different architectures and use cases, making a first step towards a systematic approach.

Moreover, part of this paper is devoted to present a frontend for the Ghidra P-Code emulator. The plugin was designed to overcome our faulty board, that prevented us from running any of the challenges for the majority of the time. Our tool was specifically developed to reduce the time needed by an analyst to deal with the context for the surgical emulation of single functions, allowing a smooth interaction with the existing Ghidra environment.

The flexible approach we followed leveraging our tool led us to complete most of the proposed challenges before fixing the board; the solutions were validated once we got a working board. Therefore, we strongly believe it is worth describing the ideas behind our solutions.

**Index Terms**—RFID, firmware, security, reverse engineering

## I. INTRODUCTION

In our society, the presence of embedded devices is nowadays prominent. Those systems are widely used, from consumer services like smart homes to critical industrial infrastructures. On those devices run software so there exists vulnerabilities and exploits. The more the number of embedded systems increase exponentially, the more our society depends on them, the more we need to secure them.

The *CSAW Embedded Security Challenge* [1] — an educational, research-oriented tournament — involves young hackers from all around the world exploring the weaknesses of the embedded systems for the last 12 years. The topic of this edition is Radio Frequency Identification (RFID) [2], a security measure used everywhere and a hot topic in the IoT

world. All the top teams, after the qualification round, faced a custom PCB and RFID firmware to solve a set of challenges revolving around the use of RFID, with reverse engineering and exploitation efforts. We, the TRX<sup>1</sup> team of students from Sapienza University of Rome, will present our solving process and techniques used in this year competition.

Firmware reverse engineering is the fundamental step to deepen the understanding of the functionalities of an IoT device. Being a difficult task, there are no fixed steps to perform the analysis, but a set of heuristics can be followed to ease reaching the proposed goal.

After mentioning the structure of the challenges, we will present the design of our frontend to the NSA’s Ghidra [3] P-Code emulator [4] that we developed to specifically solve the provided challenges. This tool can easily execute arbitrary binary code of the two involved architectures with a low requirement in terms of time to setup, reasonable execution speed and an user-friendly interface. Even if we initially solved the challenges using exclusively our tool, due to a faulty board, we believe the combination of the emulator with access to the target hardware makes the challenges far more doable.

Then, we will shortly explore the fundamental ideas behind the solutions for each of the binaries provided in the *Embedded Security Challenge*. Starting from a shallow view of the binary, analysing its behaviour, we will show how to investigate the characteristics of the target making use of the NSA’s Ghidra Software Reverse Engineering (SRE) tool and our frontend to the Ghidra P-Code emulator.

In the end we will discuss the overall experience and propose some future directions for the development of our tool — that we plan to release as an open source project after the competition.

<sup>1</sup><https://theromanxpl0it.github.io/about.html>

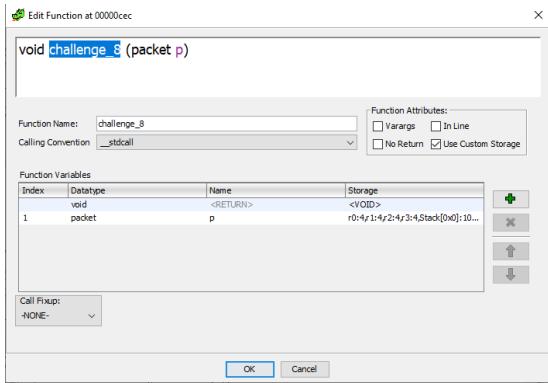


Fig. 1: Ghidra Edit function window with Use Custom Storage.

## II. PREPARING THE CHALLENGES

There are 6 challenge sets, each containing several challenges.

All the challenge binaries have debug symbols to help with the reverse engineering process.

### A. Setting the correct function parameter type

All the challenge functions are named challenge\_n, where n is the number of the challenge.

Following the references to the challenge functions we derive the following call chain:

```
receiveEvent, startChallenge, challenge_n
```

Looking at the call to startChallenge in function receiveEvent we understand that startChallenge takes only one parameter of type packet

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

Since this struct does not fit the registers designed to hold the parameters of a call, part of the structure spills on the stack.

We can make Ghidra understand this split structure by editing the function signature of startChallenge and challenge\_n adding a single parameter of type packet and toggling Use Custom Storage. After toggling Use Custom Storage (Fig. 1) we can control where every byte of the parameters is stored. The correct values are the first 16 bytes in registers r0–r3 and the remaining bytes on the stack (Fig. 2).

Although this structure split is supported by Ghidra, we observed several issues with the decompiled code. We believe that this is caused by the lack of robust support for this non-uniform storage and the reuse of the space reserved to the

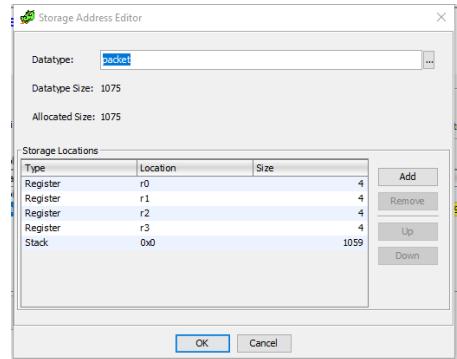


Fig. 2: Ghidra Custom Storage Address editor set with the correct values.

parameters — in particular of the registers — in the analysis engine.

So, in some instances, redefining the function signature to 4 integer parameters (stored in the registers) and a 0x423 long byte array stored on the stack helped producing a more readable decompiled code.

## III. EMULATOR

### A. Motivation

There are several aspects that make reverse engineering work challenging for firmwares:

- the firmware under examination may need to communicate with several devices (in our case we had three different components).
- the CPU architecture is different from the usual x86\_64 that is present on consumer PCs, so tools to disassemble or decompile those architecture may be less mature.
- debugging often requires additional specialised hardware.

We experienced all of the above cited problems and, additionally, the board RFID module that we received was faulty, so we did not have a reference hardware to test our findings.

At first we decided to proceed only using static analysis, in order to start working on the challenges as soon as possible, but we quickly hit some road blocks.

Since the shipment of a replacement module was taking some time, we decided to use our previous experience building Software Reverse Engineering (SRE) tools: we created a frontend for the emulator shipped with Ghidra to facilitate our analysis.

### B. Baseline

We choose the Ghidra P-Code emulator instead of other well-known projects — like QEMU [5], Unicorn Engine [6] or Uservcorn [7] — after evaluating the features and problems of those other existing frameworks.

A good emulator must have a few fundamental properties. Above all is the correctness: the same code ran inside the emulator and on a native platform must yield the same results. Completeness is also a key property, as there should not be code that just does not run inside the emulator. Last but not

least, speed is an important factor and, in our case, it must be suitable for interactive usage.

Most of the existing emulation framework are fast enough for interactive use and many fare well on the correctness and completeness side with regards to the ARM CPU architecture.

However, the success of a reverse engineering tool is not determined solely by these hard requirements: we must also consider ease of use.

An easy to use emulator and emulator frontend should:

- be easy to install.
- require little time to configure.
- not require to compile a program for each new fragment of code that we want to emulate.
- not require to write boiler plate code.
- make use of the results of the analyses and the annotation of the reverse engineer.
- offer familiar debugging commands.
- offer interactive usage.
- support many architectures.

Table I summarises the usability of the emulation frameworks, cited above, that we considered. Since P-code emulation met most of the usability requirements, we used it as our starting point.

	Unicorn	Usercorn	QEMU	P-Code Emulator
self contained				✓
Ghidra integration	basic			basic
configuration time	average	average	long	average
installation time	average	average	long	short
interactivity			✓	
number of supported architectures	average	average	high	high

TABLE I: Comparison of the usability of existing emulation frameworks.

### C. High level description

Being unable to run or debug any of the challenges due to the faulty board, our objective was to create an easy-to-use interface that allowed us to avoid spending lot of time preparing the context needed to emulate small pieces of code, mainly single functions. Manually setting registers and memory locations needed by a function to execute can be difficult when you are not familiar with the underlying assembly. In general, when you are working at decompiler level, you do not want to come back to the realm of assembly.

The main objective of the emulator front-end plugin is to offer an easy-to-use interface for every reverse engineering task that involves a function as a starting point; moreover, the plugin offers debugging and small scale fuzzing capabilities. The advantages of using the Ghidra emulator include the transparent support for source-code level emulation for all the architectures natively managed by Ghidra.

The tool enhances the normal debugging experience by leveraging the type and name information for the variables extracted by Ghidra analyses and manual annotation, offering features that are normally available only for source level debugging: printing the value of variables taking types into account, stepping to the next source line, setting the parameters for a function call (without having to manually arrange the layout in memory according the calling convention specified by the particular application binary interface).

With regards to setting the parameters for a function call, the plugin supports the use of wildcard bytes: when the plugin detects the presence of such wildcards, it will execute the function with every possible assignment for those bytes and log the execution traces to a file.

In order to minimise the need to repeat certain actions over and over, the plugin remember the last used value for each parameter, in the context of a particular function, and it also remembers the last command (or series of commands) issued during a debugging session.

The debugger also supports issuing batch commands, a very powerful feature when combined with the wildcard bytes. A common pattern found during the reverse engineering work is the *validator* pattern: a function that compute a validity flag from the input and take different action depending on the value of such flag. One example could be the challenges that were provided during CSAW ESC, in which the `challengeHash` global variable was overwritten based on the correctness of the input. Using batch commands with wildcards it is possible to set a break point right after the validity flag computation, print its value and then stop the execution; then, looking at the execution trace files we can search for a valid assignment of the wild card bytes producing the required validity.

The main contributions of our Ghidra plugin are:

- simple installation by downloading a single python script with no external dependencies.
- leverage the decompiler to simulate function calls (Fig. 3).
- highlight the position of the instruction pointer in both the decompiled and disassembly view.
- allow variable, address and register inspection (Fig. 4) and overwrite.
- support basic debugging commands (breakpoints, continue, step into, step over, print, ...).

A caveat of our work is that P-code emulation in Ghidra is still in an early stage, especially with regards to *custom operations* — P-code operations that have a special semantics that is not specified. For example, the compiler optimised `strlen` function contained in the challenge binaries uses instructions that cause an infinite loop of Ghidra P-code emulation. A simple solution to avoid such condition is to patch the affected function with simpler deoptimised versions.

### D. Fixing the board

As mentioned before, during the first weeks of the reverse engineering phase we were not able to check our solutions

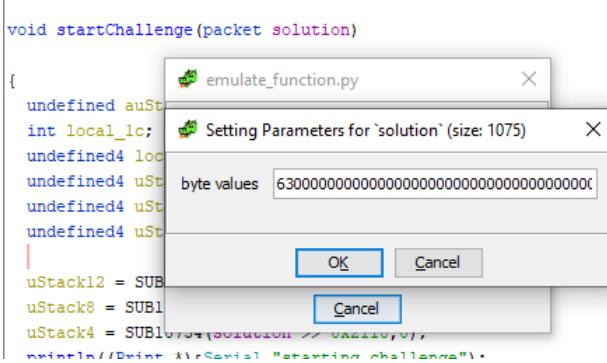


Fig. 3: For each parameter, the emulator asks for a byte string with the appropriate size.

Fig. 4: The emulator shows the value of local and global variables that are used in the function.

because the board was not working properly: the RFID tokens were not read by the board.

At first we thought it was due to the RFID device being too close to the central board, in such a way that the microcontroller was creating interference. We tried to desolder the two components, using a soldering iron and a desoldering pump, and connect them with more appropriate jumpers, but the RFID component was still not working properly.

We asked the committee for a new reader, and by replacing it we managed to get the board working and started validating all the solutions found with the emulator.

#### IV. CHALLENGE SET A

AQ-LOUNGE

The bytes in positions 76,77 are used to produce two integers —  $a$  and  $b$  — then casted to float and used for some more complicated computations.

Using the backward and forward slice analyses of Ghidra we discovered that the only check that affects the result is that  $a \times b = 0x18af$ . Therefore, we only need to find two bytes that will produce the correct  $a$  and  $b$ : since the search space is small an exhaustive search would be feasible. We decided to analyse the algorithm in a bottom up fashion and found the correct input pair 112, 232.

sender.py hash: 643a6fa20b171fdf3a9e7e1975ce62892fde9cecf2056a73d85fa2d0802d3000.

A1-CLOSET

In this challenge is very important to work with decompiled code and assembly side by side, since Ghidra has a hard time to reconcile various stack variables.

It builds a vector A of 24 elements, taking the first  $0x70 - 0x5c = 20$  elements from `rfid[0x5c:0x70-1]`, and the last 4 form `rfid[0x80:0x84-1]`. Then, the key `ESC19-rocks!` is compared against `A[A[i]]` for  $0 \leq i < 0xc$ . So we need to use the first 12 position to index the second half of the vector, which will contain a copy of the key. The easiest way to do it is to put 12, ..., 23 plus the key minus the last 4 chars in the first offset (`0x5c`) and the last 4 bytes of the key to the second one (`0x80`), but note that any permutation of the indices (first half of the vector) and corresponding char of the key (second half) would have worked as well.

sender.py hash: 8425ad5e0454e8f2398aa8a2b4a  
361e5670339dad91b5d81aef88fd940d7bac9.

A2-CAFE

The challenge reads 14 bytes from the RFID and use them to reconstruct the missing bytes in the correct output string. It starts by xoring `output[0x0:0x5]` with the key `h4x0R2`, then it sum or subtract the result of this operation with the bytes read from `rfid[0x4e:0x53]`. In order to choose if sum or subtract it uses the byte `rfid[0x5a]` as a bitmap: if the bit value is 1 then the bytes are added to each other, otherwise they are subtracted. The same operations are done with `output[0x11:0x16]` using `dr00L` as key, `rfid[0x5b]` as bitmap and `rfid[0x54:0x59]` as bytes to be added or subtracted. To calculate the correct input we need to provide in order to solve the challenge, we need to simply invert the algorithm, starting from the initial output, xoring it with the same keys and the calculating the difference between the result of the xor and the target output, if the difference is greater than zero than that value must be added so we set the corresponding bit in the bitmap and the difference if our input, otherwise it must be subtracted so our input would be the absolute value of the difference.

sender.py hash: d05235d380e913b5625d653c555  
de8925f249838896651a95bb35ea4e7863a5e

A 3-STAIRS

The challenge\_3 function reads 12 bytes from the RFID starting at offset 64, it xors them with 0xF and then checks whether or not the result is equal to the string ESC19-rocks!. Calculating the correct input is all about reversing the xor operations thus, xoring the string above with 0xF gives us the final result.

sender.py hash: 396f4b1cdf1cc2e7680f2a8716a  
18c887cd489e12232e75b6810e9d5e91426c7

## V CHALLENGE SET B

B4-MOBILE

The challenge\_4 function reads 30 bytes from the RFID starting at offset 116 and extracts from those bytes some

indexes used to access a matrix of lowercase letters. It tries to complete the target string part that is composed by only empty spaces by replacing those with the letters taken from the matrix using the extracted indexes. So, we need to provide a sequence of numbers that, when given to the checker, generates the string "challenge". The python scripts that generate the final input is the following:

```

coordinates = [(1, 2), (3, 1), (1, 0),
               (4, 2), (4, 2), (2, 1),
               (5, 1), (3, 0), (2, 1)]
inp = []
for (a, b) in coordinates:
    for _ in range(b + 1):
        inp.append(a)
    inp.append(0)
for i in range(len(inp), 60):
    inp.append(0)
for i in range(0, 60, 2):
    x = (inp[i] << 4) | inp[i + 1]
    print("RFID[132+{:d}]={:s} ".format(i // 2, hex(x)))

```

sender.py hash: f2f3792453040e837e7e1584e72  
859bfaa6b8c09d73d185be53b35886b6455c2.

#### B5-DANCE

The challenge reads 8 bytes from the RFID (from the offset 147), computes the SHA256 of this string and checks if it matches a hard-coded hash. A quick search of the hash reveals that the string `password` produces such hash, and in fact putting the corresponding bytes at the right offset solves the challenge.

sender.py hash: e631b32e3e493c51e5c2b22d148  
6d401c76ac83e3910566924bcc51b2157c837.

#### B6-CODE

The `challenge_6` function reads 1 byte from the RFID at offset 155. It calculates the hash of the string `imjustrandomdatathathasnomeaningwhatsoever!` with the input byte appended at the end, and then compute a hash and checks whether it is equal to a hard-coded one. If we check the initialisation values of the hash function and the length of the output we can see that the function is MD5. So, in order to solve the challenge we need to find the right byte that, when appended to the string above, passes the check, and this can be easily brute forced.

sender.py hash: 372ded6746e45ef7c8ad5a22c57  
38a4b5aa982da66bc8a426aa1cca830d05af3.

#### B7-BLUE

The challenge extracts 8 bytes from the RFID from the block 0x3C, hashes them (using SHA256) and compares them with a fixed value. The hint in the challenge suggests using the blue tag, and reading such tag with an RFID reader reveals that in the block 0x3C there are the bytes 5D:92:95:A4:B9:80:18:B5 whose hash value perfectly matches the required one.

However providing the blue tag to the challenge still makes it fail. Since there seems to be no more logic to be analysed in the Teensy, we moved into analysing the operations performed on the ATMega side.

We discovered that the ATMega reads the RFID card using default keys FF:FF:FF:FF:FF:FF to authenticate for all the challenges, except this one, where the custom key 5d:6a:XX:64:f5:91 is used to read data in the blocks 0x3C:0x3F for this challenge. XX is set by default to 00 and depends on the value of the `key[2]` field of the global `packet` variable `w`.

The lack of just a single byte in the key allows us to extract the key for the block 0x3C-0x3F in the blue tag using a brute-force attack with *mifare tools*: 5d:6a:E4:64:f5:91. Therefore setting XX to E4 would make the challenge succeed and accept the blue tag.

Since XX is initially set to 00, we can write on the RFID card or the blue tag using custom tools — without using the provided sender, since it does not allow us to override the keys in the sectors — and setting the block 0x3C of the card to 5D:92:95:A4:B9:80:18:B5 and the `keyA` in block 0x3F to 5d:6a:00:64:f5:91 produce a valid card that is accepted from the challenge.

In order to solve the challenge using the provided sender, we found out that the Teensy board supports other commands beside '`p`' to program a card. What we need is the command '`a`' (authenticate) which instructs the AtMega to update the key field of the `packet` variable `w`.

Sending the command '`a`' with `k[2] == 0xE4` allows us to use the blue tag to solve the challenge.

There are two possible pitfalls in this challenge:

- the AtMega takes some time to receive the '`a`' command, so it is helpful to insert a delay in the sender.
- the key update does not survive a reset, so it is important to not reset the board before checking the solution.

sender.py hash: ae1eb10d1025f4a7ff055447b52  
08a36412318fc015ebb325deb78806a7e96bf.

## VI. CHALLENGE SET C

#### C8-UNO

We found a base64 encoded hint that ask us if one instruction set computers are 1337. In fact, the OISC implemented in the challenge is a version of `subleq` [8] (subtract and branch if less or equal), and in order to solve the task we need to provide a `subleq` program that generates the correct output string.

Before digging into the actual program structure we have to check if we have some constraints to consider: the maximum length of our program is 48 bytes, and considering that each `subleq` instruction is composed of 3 bytes, it means that we have at most 16 instructions; the interpreter is made in such a way that after reading 30 bytes (size of the output), it will execute all the remaining instructions until the end or until it encounters an exit instruction, so we need to consider this if we want to implement a kind of loop.

The final `subleq` program is the following:

```
[0] = (1, 1, 34)
[1:33] = "jihgfedcba onu egnellahc devlos"
[34] = (50, 40, 49)
```

```
[37] = (42, 40, 49)
[40] = (33, -1, -2)
[43] = (0, 40, 49)
[46] = (51, 51, 34)
[49] = (-1, 2, any)
```

The first instruction simply jumps to the offset 34, and in between these two instructions there is the reversed target string. Then, we implemented a loop that starting from offset 33 calls the read primitive in order to give the right byte to the program, decrements by one the index and then iterates in order to print the whole reversed target in the right order. To make the program stop when 30 bytes are read and avoid unwanted behaviour, at each iteration we subtract 2 from the index and check whether or not it is equal to 0: if so, we go to offset 49 which is our exit instruction, otherwise it add back 2 and continue iterating.

```
sender.py hash: 2de37e3efff64444827bb440722
bb8781f459203b3d61d065540e7750c741dad.
```

### C9-GAME

The challenge plays a *Tic Tac Toe* game, starting with a partially filled board, and we need to specify our moves (starting from the offset 156) in order to win (impossible) or tie the game. Since we are the first to move and all the moves are forced (expect for our last one, but it doesn't matter at that point), we just need to provide one of the two possible correct sequences that cause a tie.

```
sender.py hash: 63c0b41f89bbf493ba791c092b3
e5473e243b9c16666f1e5eaa82bc52eeb1613.
```

### C10-BREAK

The challenge computes a fixed checksum (0x4161) and asks for two bytes in the RFID that, together with the values of buttons, matches that checksum when composed in a fixed way. Only a subset of the button values admits a solution ( $A = 0, B = 0$  being one of them), and there exist two values (65, 97) for the two RFID bytes that work for all such button values, so we used that. Given the small search space, we found those values by simple enumeration of the 3 bytes (buttons plus the two bytes).

```
sender.py hash: 2d3448f09329f453e6f3a5403d8
9c061a9dabfb9103ad6b8cc86d16345a7547.
```

### C11-RECESS

The challenge reads 4 bytes from `rfid[0xa1:0xa4]`, generates a Cyclic Redundancy Check (CRC) of them using a custom implementation and checks the result against 0x36476684. To solve this, we have to re-write the custom crc function and write a simple script that can bruteforce, in parallel, the 4 bytes needed to pass the check. Instead of losing too much time setting up a multi-threaded program, we ran 4 instances of the same script each one starting bruteforcing from different offsets: with this approach in less than a minute we got the correct bytes sequence.

```
sender.py hash: 370815b8d8fde829f5c35f893d0
b4139d61a775baa4181fcac1ffe014bde9ea.
```

## VII. CHALLENGE SET D

### D8-BOUNCE

The challenge reads 47 bytes from the RFID, in a buffer of 8 bytes, writing a byte or not in the stack based on a bitmap we can define in the RFID itself. Therefore we can effectively overwrite the return address to control the execution flow.

However, the challenge enters an infinite loop instead of returning, if some condition is not met: the value of the variable `c`, initialised as the xor of the last two bytes of the return address (identified as `var_s2` and `var_s5`) has to match the value of the xor between the value of buttons (`A=0x1` and `B=0xd`) and the last byte of the return address.

Since we can control all of these values, while overriding the stack buffer, it is straightforward to write a valid return address where to jump to. The problem is which address, since the challenge function does not seem to include any code dealing with challenge success.

However, we can notice that the challenge sets the variable `use` before returning. Examining the references to this variable we discovered the `fillChallengeHash()` function that makes the challenge succeed if such variable is set.

Therefore, setting the RFID bitmap and values to overwrite the variables to return to the `fillChallengeHash()` function at address 0x71C (0x71D to preserve thumb mode) solves the challenge.

Initially our payload did not take thumb mode into consideration, therefore it did not work on the board. Using the emulator we were able to set breakpoint at controlflow changing points and then single step the execution. Using the emulator we were able to determine that even though the value of the return address was correctly modified, upon stepping after the end of `challenge_8` the emulator would raise an error because it would find an illegal instruction. Comparing the execution trace with a modified return address with a normal execution trace we were able to observe the difference in the alignment of the return addresses, thus confirming the switch to thumb mode.

```
sender.py hash: cf50713a6da3ddbd22a55ca4e70
6111c610bc192a928e080eb923dcf8a5444a5.
```

## VIII. CHALLENGE SET E

### E12-STEEL

We found a hint in a `base64` referenced in the challenge function that asked whether repeated hashing would be more secure.

The challenge then proceeds to compute a hash function 10 times, each time feeding the result of the previous hash. The challenges is solved only if the resulting hash matches a constant embedded in the firmware.

The challenge only uses 3 bytes from the RFID card so the key solution is already small, moreover the three bytes are reduced to a single byte value that is fed to the hash function.

By looking at the decompiled code we can confirm that the hash function, identified by Ghidra as H45H, is a standard MD5 hash; we confirmed this suspicion by emulating said function

with our emulator by fixing the input value and comparing the result with the expected hash.

We first tried to rewrite the logic in python and then in C, however we could not find any value that, after 10 iterations, had a final hash value matching the hash of the solution of the challenge.

Since the rewriting was not successful, we decided to use the fuzzing capabilities of our emulator to try all the possible byte values and looked at the execution traces for the correct answer.

The emulator found the correct answer after a few minutes.

```
sender.py hash: 0f78e37b899630154e306dbb588  
8e27f8bb7ce431ea652aea693918e490d072c.
```

#### E13-CAESER

The challenge uses 4 bytes from the RFID content and the buttons values to compute the `poly()` function and check its result with some fixed values. The function is called three times, and examining the function we can see that it contains simple operations in small, fixed length, loops.

These characteristics make the function easily manageable by an SMT solver (we used z3), able to find the right values to pass the check.

Leveraging the decompiled code by Ghidra, we quickly wrote a solver that reasoned on the same constraints of the challenge, able to compute the needed values in a few seconds, that can be fed to such challenge passing the checks. In particular, given the solution for the buttons value being 0xab, the final assignment for the buttons to be set manually is A = 0xA and B = 0xB.

```
sender.py hash: da3b81c1734cfb971f5b387c949  
addaa862e4bbe3427a9f2aa381d1d3afe3ebf.
```

#### E14-SPIRAL

The challenge reads 4 bytes from `RFID[397:401]` and uses them to create two short (2 bytes integers); then, it applies a series of sum, shifts and xor to these two values and finally checks that the results are equal to two constants 0xa29f and 0xd481.

Since the solution space is 4 bytes and we have two 2-bytes checks, we expect that there will be only one valid assignment for `RFID[397:401]`.

Due to the small solution space, we can easily write a C program using Ghidra decompiled code and try all the possible values of `RFID[397:401]`.

The solver takes just 5 seconds to find the correct values (0xCAFE 0xFADE) and saved us the time needed to invert the transformations.

```
sender.py hash: 26ee8470c732dfc821bbe0561b4  
46dc8086560e4e222b22e6a74e559d90a7d61.
```

#### E15-TOWER

The challenge computes the hash of 13 bytes taken from the RFID and compares them with a fixed value. The hash function used is SHA256, despite the fact the function is effectively called `blake_256`. Obviously it would be very

difficult to brute-force 13 bytes to match the hash, without making assumptions on the input, and the value provided doesn't seem to be a known hash.

Strangely, the hint given in the challenge is represented by a variable called `bd11882364` with the value `ht` encoded in base64. This rang a bell, since in all the other challenge the variable containing the hint was called simply `hint`. Examining the debugging information we can spot multiple variables with names starting with `bd`, representing base64 encoded strings. Extracting the `dwarf` from the elf, we can collect all the variables with such characteristic, along with the places they are used. Examining each of these places we can extract all the base64 string generated: decoding and concatenating the results we obtain the pastebin url <https://pastebin.com/VegeWmJP>, and the paste contains the string producing the right hash: `ndixlelxivnwl`.

```
sender.py hash: b019c48299dd33ec6fdc94da9d5  
ad06018549ee58f4a829a44d15e6980c22cbb.
```

## IX. CHALLENGE SET F

#### F17-SPIRE

The challenge reads 8 bytes from the offset 640 of the `RFID` field of the packet into a 4-byte buffer. This allows us to override the `len` variable adjacent to the end of such buffer. That variable controls the number of iterations in the subsequent `for` loop that reads the bytes from the offset 719 (going upwards) of the `RFID` if the byte at offset 879 has the last bit set. Using the second `for` loop we therefore are able to override multiple variables on the stack. The goal is to override the `privilege` variable with a positive integer that makes the challenge succeed. Such variable is 12 bytes upwards with respect to the start of the second array used in the second `for` loop. To reach the variable we have also to override a pointer dereferenced in the function with a valid address. We chose the address of `__malloc_sbrk_base` at `0x1FFF9894` writing a value of `0xffffffffffff`. Placing the value `0x1` in the `privilege` variables, solves the challenge.

```
sender.py hash: bdc2d12ebf5a9c9de0b2028a4be  
dc13145b88bc9a1c08ec09728aab3e48b2780.
```

## X. DISCUSSION

The reverse engineering tasks proposed in this challenge prompted us to study how the involved hardware worked and interacted both with different devices and with the software that we were reversing. We presented how we solved all the proposed problems and the design ideas behind the development of a frontend for the Ghidra P-Code emulator to easily deal with such challenges. Not only our knowledge in the field increased a lot solving those challenges, but also we now have a reusable tool for the future challenges.

To facilitate analyses and extensions of our tool, the source code of the plugin will be made available at:

<https://github.com/TheRomanXpl0it/ghidra-emulate-function>

### A. Future Directions

Being a tool initially developed with the goal of easing solving those challenges, there are multiple directions towards its evolution to make it a more general purpose solution and overcome some of its limitations. One weakness is the support for system call: currently the analyst has to identify all the sources of input and use the debugger commands to compensate the lack of support for such calls. One way to move forward in this direction is to combine hardware debuggers and the emulator using framework such as Avatar2 [9].

Another aspect that in our opinion needs more investigation is a tighter integration with Ghidra decompilation process. Currently, the type information is used only to get the size in bytes of a variable and to pretty print strings and numbers; adding additional formatters, as found in source level debuggers as GDB, would be a great addition.

A direction that was not explored in this work is the addition of reversible debugging to the emulator. At present, we only log the *addresses* modified during the emulation and not their content. It would possible to offer reversible capabilities similar to rr [10], GDB [11], WinDBG [12] and QIRA [13], as an emulator has complete knowledge of all the state changing operations that happen, and the execution is always deterministic.

### ACKNOWLEDGEMENTS

We would like to thank Andrea Fioraldi and Pietro Borrello for their valuable feedbacks on the final report.

### REFERENCES

- [1] “Cshaw embedded security challenge,” 2019. [Online]. Available: <https://cshaw.engineering.nyu.edu/esc>
- [2] M. Bolic, D. Simplot-Ryl, and I. Stojmenovic, *RFID Systems: Research Trends and Challenges*, 1st ed. Wiley Publishing, 2010.
- [3] “Ghidra,” <https://ghidra-sre.org/>, 2019, [Online; accessed 1-Nov-2019].
- [4] “Ghidra documentation: ghidra.pcode.emulate.” 2019. [Online]. Available: [https://ghidra.re/ghidra\\_docs/api/ghidra/pcode/emulate/package-summary.html](https://ghidra.re/ghidra_docs/api/ghidra/pcode/emulate/package-summary.html)
- [5] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247360.1247401>
- [6] A. Q. Nguyen and H. V. Dang, “Unicorn: Next generation cpu emulator framework,” 2019. [Online]. Available: <http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>
- [7] lunixbochs, “Usercorn analysis and emulator framework,” 2019. [Online]. Available: <https://usercorn.party/>
- [8] P. J. Nürnberg, U. K. Wiil, and D. L. Hicks, “A grand unified theory for structural computing,” in *Metainformatics*, D. L. Hicks, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–16.
- [9] M. Muñch, D. Nisi, A. Francillon, and D. Balzarotti, “Avatar2: A multi-target orchestration platform,” 2018.
- [10] “Mozilla rr tool.” [Online]. Available: <https://rr-project.org/>
- [11] GDB v7, Tech. Rep. [Online]. Available: <http://www.gnu.org/software/gdb/news/reversible.html>.
- [12] “Windbg time travel debugging documentation.” [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview>
- [13] “Qira home page.” [Online]. Available: <https://qira.me/>