Team TRX Sapienza
CSAW ESC 2020 Final Report

Matteo Almanza
Computer Science Department
Sapienza University of Rome
almanza@di.uniroma1.it

Cristian Assaiante
Computer Engineering Department
Sapienza University of Rome
assaiante.1744195@studenti.uniroma1.it

Qian Matteo, Chen
Computer Science Department
Sapienza University of Rome
chq.matteo@gmail.com

Dario Petrillo
Computer Engineering Department
Sapienza University of Rome
dario.pk1@gmail.com

Leonardo Querzoni
Computer Engineering Department
Sapienza University of Rome
querzoni@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 RISC-V challenge binaries provided in the Embedded Security Challenge 2020. Our approach is general enough to be extended to different architectures and use cases, making a first step towards a systematic approach.

In particular, part of this paper is devoted to present the improvements we did on last year version of our frontend for the Ghidra P-Code emulator – plugin initially designed for the ESC 2019. We specifically developed our tool to reduce the time needed to surgically emulate single functions, reducing the time spent to deal with the surrounding context, allowing a smooth interaction with the existing Ghidra environment. The approach was combined with binary patching to modify and study the behavior of hard-to-understand binaries and undocumented features.

The flexible approach we followed leveraging our tools led us to complete all of the proposed challenges. Therefore, we strongly believe it is worth describing the ideas behind our solutions.

***Index Terms*—RISC-V, IoT, reverse engineering, firmware security, WiFi access point**

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 13 years. The topic of this edition is hacking the firmware of a wifi access point running on a RISC-V IoT platform, a hot topic in the IoT world.

All the top teams, after the qualification round, faced a custom RISC-V firmware to solve a set of challenges revolving around the reverse engineering of unkown firmware. We, the TRX [1] 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.

At first, we will present how we improved our frontend to the NSA's Ghidra [2] P-Code emulator [3] that we developed last year to specifically solve the ESC 2019 challenges. This tool can easily execute arbitrary binary code of the 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 the travel restrictions that prevented our team to meet in-person, we believe the combination of the emulator with access to the target hardware makes the challenges far more attainable.

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 it's 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 is already released as an open source project and used by people of the security community during this year, but will be integrated with our improvements after the competition.

[1] https://theromanxpl0it.github.io/about.html

II. Improvements to the Emulator

This year we could not meet in person to work on the challenges. We also live in different cities, so we could not easily move the challenge board around.

It was therefore critical to reduce the reliance on the hardware when working on the challenges.

The solution we adopted is based on the work [4] [5] we did for CSAW ESC 2019: our Ghidra P-Code emulator.

We improved the emulator in several ways:

- Improved support for hooks
- Better support for code coverage
- Improved scripting support

Notably, there was no need for improvements to support the new architecture proposed this year, RISC-V, as our tool can emulate all the architectures supported natively by Ghidra or by third party Ghidra plugins from the first day — unlike similar approaches like [6] [7].

A. Hooks

Our tool allows us to replace part of the emulated binary using high level python code.

As an example, let's consider a replacement implementation for `memset`:

```
@hooks.args
def memset(s, c, n):
    for i in range(n):
        s[i] = chr(c)
    return
```

Here `@hooks.args` is a python decorator that takes care of extracting the function signature and calling convention from Ghidra and takes care of reading the parameters from the registers and memory in the emulator, pass the values to the python code and then writing the return value and memory writes caused by the python implementation back into the emulator.

Other frameworks for emulation usually do not have this level of integration with reverse engineering tools, so the user needs to reason at a much lower level when writing hooks (i.e. manually computing offsets on the stack, use low level APIs of the emulator to read and write registers and addresses in memory).

This was particularly useful to easily replace the functions that rely on the hardware to work correctly. We implemented a subset of those functions and this allowed us to emulate the challenges reliably.

B. Code coverage

Our extension allows us to view the code coverage of the current and past executions of the emulator, making it easier to visually inspect how different parameters affect the execution of a program.

Using another extension we developed we can see the coverage information on the function call graph.

Additionally our extension can export the coverage information into standard formats, which enables the use of extensions designed for differential debugging such as [8].

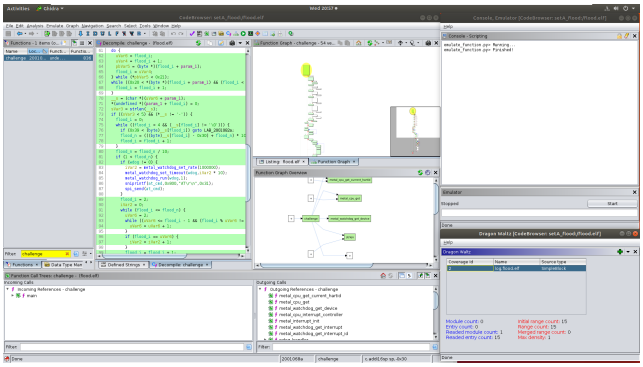

Fig. 1: With our extension is possible to visualize the trace of an emulated function in the decompiled code view, disassembly view and also on a function call graph

C. Scripting Support

Last year we supported basic scripting of the emulator by providing a list of commands for the emulator.

For this year's version we added more commands to cover common use case and also support for python code in commands for more automated emulation sessions.

The addition of python code simplified scripting for challenges where a moderate amount of fuzzing or bruteforce was needed.

III. The HiFive1 Board

The HiFive1 board [9] provided for the competition is an IoT development board based on the Arduino form factor, which combines a 320MHz RISC-V microprocessor with an ESP32 module and an integrated debugger. The board also offers an RGB LED, which is used by the ESC challenges as a status indicator.

IV. Challenge Set A

A - AMNESIA

This challenge repeatedly takes a string and converts it into a yodel. It then saves the resulting yodel in a section of flash memory alongside a checksum.

The challenge will print the flag if it finds certain characters in the same section of flash memory and a valid checksum at the end. The challenge does this check at the start of each round, before writing the yodel to the flash.

Normally those characters cannot be written since yodels have a very limited character set, however the binary puts these values after the check, before being overwritten shortly after with the yodel.

The idea is that, since those values are written into the flash memory, they are going to persist across reboots, therefore we can do a timely reset of the board and pass the check when the board restarts.

Luckily, it is possible to time this process by watching the board: a led will briefly turn green when the values are saved into memory.

There are many input strings that will solve the challenge. We used: `'A' * 60` which results in `<AAAAAAAAAAAA*AAAAAAAAAAAA0200.*AAAAAAAAAAAAAAAAAAAAAAAAAAAAah9c` to be written in flash memory.

A - BREAKFAST

This challenge contains a global array of 60 permutations of the string `"aaabnn"`, which is used to encrypt the provided key. It accepts an input string and expects it to be in the `<number><space><key>` format. It then takes the number, which we will call R, and rotates the global array left by $(R \mod 59) + 1$ positions. For each character ch in the key, $ch - 0x41$ is used as an index into the global array to obtain a 6 character string which is concatenated to the encrypted output. Finally, if the output matches a fixed string `abana...naba`, the key is accepted.

This is a weak encryption function, and in fact it is easily inverted by the following process: for each 6 character chunk of the target string, find its index into the array, add $0x41$ to it and append this number (converted to character) to the recovered key. Since the challenge offers multiple solutions, one for each possible rotation of the array, we decided to use `"4 ItsNotBaconian"`. As a result, the board prints the success message `"Correct!! You won 100 points!!"`

A - BURST

We approached this challenge as a classical crackme task. It expects to find three numbers a, b, c in the input string, which are all 32 bit unsigned integers. In order to complete the challenge, the following conditions must hold:

- $(a - 0x3b3440) < 0x3e9$
- $(((b - 1 << 1) + 1) * a + c == 0$
- $a, b, c \neq 0$
- a, b, c must be $7, 4, 7$ characters long, respectively

We used the Z3 theorem prover [10] to quickly find a solution for the given conditions. One such solution is

$a = 3880128, b = 2212, c = 3545728$

After sending the correct input string, `"3880128 2212 3545728"` to the board, we received the challenge completion message `"Correct!! You won 50 points!!"`

A - FLOOD

In this challenge we have a service that computes the number of prime numbers lower that our supplied input.

The service installs an interrupt handler for timeout events that happen while it is computing the number of primes. In this interrupt handler the binary sets a global variable to zero (while normally it will be greater than 1) and the service will print the flag if that variable is less than 2.

The challenge checks that there are only 4 number like digits in our input, so, under normal conditions, it will terminate the computations before the timeout.

The core idea is to exploit an error in the number parsing code of the service as it allows ascii character between '!' and '0'. Using non digit characters results in a numeric underflow that makes the service parse the input as a very large number.

Thus sending '!!!!' as input will result in a timeout. In the interrupt handler the service checks how many character can still be read from the input, so if we send a large number of characters after '!!!!' we can fulfill all the winning conditions of the challenge.

Flag `*SaveforReport?TarbelaItaipu?Hoover?AtatuRK`

A - PARTHENON

This challenge takes a string in input, applies a custom `UD5` hash to it and checks whether the output matches a constant string. At startup, it also prints an encrypted string: `"etemsletryaetyscrsemomtaahnuonlhciiieutlalcpsc"`

At first, we approached this challenge as finding a preimage for the provided UD5 hash. It quickly became obvious that this wouldn't be an easy task and would take up a lot of our time, so we moved onto trying to decipher the string. After an online search, we identified the encryption algorithm to be a columnar transposition cipher [11], a special type of the classical transposition cipher. This cipher, like most classical ones, is vulnerable to statistical attacks, so we used an online tool to recover the encryption key and decrypt the string. The key is `"bdeca"` and the decrypted string is `"youmeantotellmethatmyclassicalcipherisntsecure"`.

By sending the decrypted string to the board, we solved the challenge and received the following success message: `"Correct!! You won 70 points!!"`

V. Challenge Set B

B - CHASE

This challenge is a listening server that changes its ip address and port at every new connection using a custom random number generator. At the first connection, it asks for an input `Hello` string and it outputs a sentence in a random language according to the same random number generator, the next ip address and the next port. Upon each connection after the first one, it does exactly the same thing omitting the next connection parameters After six connections in a row, the challenge is solved and the flag is given.

Our goal is to break the random number generator in order to be able to predict the next endpoint.

With the data retrieved from the first connection and a re-implementation of the generation algorithm, we are able to calculate a set of potential initial seeds by brute-forcing every possible value, looking for those that end up generating the same ip address, port and language. From the second connection onward, the only data we have is the language. We can do the same brute-force over the set calculated before looking for those seeds who generates the same language. Luckily, from the second connection the initial set of 18 potential seeds is reduced to a single one, thus we are able to predict everything we need to solve the challenge.

After the sixth connection we received the following success message: "Key:1*4mH3r3 Challenge Solved!!You won 100 points!!"

B - ESROM

The challenge asks to input a string and check whether or not it is equal to a saved hash calculated using the so called SHA2 function – which is not the standard hash function.

In case the input is not correct, the green led starts to blink in a fixed pattern which can be extracted from the binary: "000 0 10 100 2 1 0000 0 2 0010 111 0100 0100 111 011 00 10 110 2 111 0001 0 010 2 10 0 1 1010 01 1 111000 2 010 0 11 111 010 000 0 0010 001 0100". From the behaviour of the binary, we can infer the following meaning:

Symbol	Meaning
0	morse dot
1	morse dash
2	end of word
space	end of letter

The binary sleeps for $500ms$ for a 0 and $250ms$ for a 1, but the two symbols have to be swapped to get a meaningful output. Once decoded, the sequence become: "SEND THE FOLLOWING OVER NETCAT: REMORSEFUL"

Sending the string "remorseful" we get the success message: "Challenge Solved!!You won 70 points!!".

B - MIDDLEMAN

This is the solution for the first version of the challenge that was released.

We are given four message/ecdsa signature pairs. The program expects as input the hash of the signature of a given string. The hash is composed by 4 32bit integers (o_i in the pseudocode below) and it is evaluated as follows:

$$e \leftarrow [8, 9, 11, 10]$$
$$b_i \leftarrow input[i \cdot 4 \ : \ (i+1) \cdot 4]$$
$$o_i \leftarrow (SHA1(b_i))^{e_i} \ mod \ 2^{31} - 1$$

We can mount a bruteforce attack to recover each b_i separately from the corresponding o_i in a less than a minute. Due to the exponents and the modulo chosen for the modular exponentiation, we will have more than one solution. One of them is (hex encoded) "32323032203131363820313 2abf2b063". With said input, the program will give as output "Challenge Solved!!You won 150 points!!".

B - MIDDLEMAN-UPDATE

This time the hash is evaluated in such a way that it is impossible to bruteforce 4 bytes at a time, but a function added in the new binary that parses the input hints that the program expects a signature with format "$P_x \ P_y \ r \ s$", where P_x and P_y are the public key coordinates and $r, \ s$ the result of ECDSA signature. The public key is hardcoded in the binary, the r parameter depends on a nonce that is generated for every signature, but since the RNG used by the binary is faulty (it returns always 4), we can easily find out r. It is not clear how to find the s parameter, as the ECDSA needs an hash function that generates an integer smaller than the curve generator order, which in our case is 131. We tried some of the most used hash functions modulo 131 without success. Since the s parameter can only be smaller than the range of the generator, we have a few possible value for it, so we decided to enumerate all possible signatures using the emulator. We put a breakpoint in the "correct signature" basic block and check, for each signature, whether the breakpoint is triggered. The correct signature turned out to be "1341 1979 125 97". With such input, the program will again give as output "Challenge Solved!!You won 150 points!!".

B - QUIZ

The challenge presents itself as a maths game, where we are repeatedly asked to solve simple operations, such as the sum, product, sum of squares and difference of two numbers. The challenge times out after 10s, so while theoretically it could be solved just by hand we decided to automate it with a Python script. There were two main challenges which had to be addressed in the code.

The first obstacle is the format used for numbers, which are sent out by the board in plain English and not in decimal, and which we addressed by writing a parse_number function that would convert numbers back into decimal in order to perform calculations.

When testing the solution, which needs to run for a total of 9 times in a loop before the flag is found, the script would often hang or behave erratically. After a thorough analysis we traced the behaviour back to the board sending corrupted and/or duplicated output in some occasions, which we believe might be due to race conditions in the spi or networking code. We found that sending commands at a slower rate was helpful in order to receive a more stable output, so we added sleep instructions between subsequent commands and code to send our answer again after a timeout if the board did not respond.

Once we had a stable script, we run it to obtain the following output, solving the challenge:

"Correct: 24233C+50152A-
Challenge Solved!!You won 130 points!!"

B - SEQUENCE

The challenge calculates a fibonacci-like sequence at startup, and then checks that the two provided integers are equal to the values at indices 24 and 28 of this sequence. Instead of reimplementing the sequence generation function, we decided that emulation would be way faster. We used the Ghidra pcode emulator on the fibonocci function and dumped the sequence global array from memory. Using this data, we quickly obtained the required values, 0 and 38.

After sending the string "0 38", the board replies with "Challenge Solved!!You won 50 points!!", completing the challenge.

VI. Challenge Set C

C - GAME

The challenges automatically simulates a random game of black jack against of a player against a dealer. The player starts with an amount of money of 1 dollar, and every time he wins the amount gets shifted to the left by one, if it looses the amount becomes zero. When the player reaches 1000000 as the amount of money, he wins the game and the flag gets printed. The challenge uses a pseudo-random number generator to decide each card that the player and the dealer get, that is reseeded at the start of every game. The PRNG is in the function `encrypt()` and it is in the form: $rint = (rint \oplus 0x60ce) + 0x9f41$.

The only input that we can provide to the challenge is when to start playing, and luckily the seeding mechanism of the PRNG is based just on the game number and on the tick count of the CPU, which is affected by the time, and it presented below, with `rint` being the seed:

```
TVar1 = xTaskGetTickCount();
iVar2 = (TVar1 / 1000 - game * game * game) + 0x417;
rint = iVar2 - iVar2 % 3;
```

The seed is then printed at the end of the game, along with the results. To understand how the tick count of the CPU varies with respect to time, we quickly wrote a python script that measured the time at the start of different games and reversed the seed initialization to obtain the tick:

```
def rint2tick(r, game):
    game = game % 10
    res = (((r - 0x417)) + (game*game*game)) * 1000
    return res
```

Using this utility, we noticed that the tick count could be predicted easily, since it seemed to be greatly approximated by milliseconds. Therefore, we wrote a simulator of the game to reproduce the challenge and bruteforce the tick count at which the second game should have started, that would result in the player winning enough games to reach the required amount of money. The script yielded 152000.

Finally, the attack is to play a dummy game to get the seed of the first game, recording the starting time, and reversing the seed initialization to estimate the CPU tick the starting time was referring to. Then it simply wait enough milliseconds to reach the desired tick, then start a new game. We found the attack worked more reliably adding 3000 ticks to the reversed one, to compensate the `mod 3` operation at the end of the seed generation.

Performing the attack correctly the challenge prints: `Flag:D0u8l3d0x4+Wice;e2 1191 271182`. Interestingly the challenge never prints `"Challenge Solved!!You won 150 points!!"` since the `challenge()` function always returns 0.

C - HUE

The challenge asks to input either an integer or a string. If an integer is provided the challenge simply enables or disables the LEDs of the board to create the desired hue, otherwise the challenge applies a custom hash function to the input and check if it matches a 16 bytes desired result. In the case it matches it succeeds. Ignoring the integer part (even if it probably contained hints for the solution) we can focus on the custom hash. This hash function, called `SHA2`, simply divides the string in 4 equal parts and applies another custom hash function `SHA1` to each of the parts which results in 4 different 32-bit integers. Finally concatenates 4 different modular exponentiations on the 4 integers to produce the result.

To understand which are the 4 desired results of the `SHA1` function we can export the C code of the functions and bruteforce all the possible 32-bit inputs to the modular exponentiations until they match the required results. The expected results are the integer we want the `SHA1` to output, that will then match the target hash once the exponentiation is applied. Not all the results where unique: the first `SHA1` output must be one in $\{0x2582229b, 0x5a7ddd65, 0xa582229b, 0xda7ddd65\}$, the second one must be such that the lower 20 bits match $0x2c27e$ and the upper 12 bits are a power of 4, the third one must be $0x17be800b$, and the fourth one has to be a multiple of $0x10$. Encoding these constraints, we can proceed to independently bruteforce 4 different strings, that will match the desired `SHA1` in less than a minute. These results will not be unique, and limiting the search to 4 strings of 4 bytes each in the alphabet `"ABCDEFGHIJKLMNOPQRSTUVWXYZ\x00"` produces a readable string, which is the intended input: `"THINKCOLORFULLY"`. Sending that string we get the output: `"Correct!! Challenge Solved!!You won 150 points!!"`.

C - RECYCLE

The challenge uses a Linear Congruential Generator (LCG) to generate a random key and pick random words from a hardcoded wordlist and concatenates them, generating a sentence. It then outputs the ciphertext $key \oplus sentence$. We can then query other ciphertexts (with the same key but different sentence, generated using the same LCG) or submit the recovered key. Since we determined statically that the seed value was very small, we solved the challenge bruteforcing the LCG seed, generating our guess key and sentence and check whether $guess\_key \oplus guess\_sentence$ is equal to the given ciphertext. Sending the correct key back to the server gives as output `"Flag:Uo`$\hat{}$`'B< a4100511e625e15562246032139314l386e4c265a1 c0f0c75 Challenge Solved!!You won 100 points!!"`

C - VIRTUAL1

This is the first of two challenges implementing virtual machines. The challenge begins with a setup phase, in which the four available registers r_i are initialized to the fixed values 0, 1, 2, and 3; then it executes the provided instruction and checks whether the condition $r_2 == r_3$ holds.

The three supported instructions are `ADD`, `SUB`, and `MUL`, and they all accept three parameters, `dst, src1, src2`.

One possible solution is `ADD 2 2 1`, which translates to $r_2 = r_2 + r_1 = 2 + 1 = 3$

By sending that string we get the output `"Challenge Solved!!You won 50 points!!"`

C - VIRTUAL2

The challenge implements a more complex virtual machine, with support for the following 8 opcodes:

0	1	2	3	4	5	6	7	8
add	sub	mul	and	or	xor	load	store	halt

All opcodes take the same operands as the `Virtual1` challenge, and the registers are initialized in the same way. This version of the VM adds support for a memory area, which is initialized with a value of -1 in even numbered cells and -2 elsewhere. The win condition is more complex, involving both registers and memory locations: $r_2 == mem[8]$ && $mem[8] == mem[9]$.

Since we don't have a direct way to load an immediate in a register, we can use the following operations to put the value 8 in register 0 and 9 in register 1, and to then copy the values from/to memory:

Operation	State
$r_0 \leftarrow r_2 + r_3$	$r_0 = 5$
$r_0 \leftarrow r_0 + r_3$	$r_0 = 8$
$r_1 \leftarrow r_0 + r_1$	$r_1 = 9$
$r_2 \leftarrow mem[r_0]$	$r_2 = mem[8]$
$mem[r_1] \leftarrow r_2$	$mem[9] = r_2$

Once converted to the assembled form that the board expects, this is the input that we used to solve the challenge: `"0 0 2 3 0 0 0 3 0 1 0 1 6 2 0 0 7 1 2 0 8 0 0 0 "`. By sending it we get the output `"Challenge Solved!!You won 100 points!!"`

VII. Challenge Set D

D - CORRUPT

The challenge provides us with 3 files this time - an ELF file and two Intel hex files: `challenge.hex` (that matches the ELF) and `challenge.heks`. Analyzing the ELF file we can see that the challenge is quite straightforward: when the user provides any input, it tries up to 10 times to read 64 bytes from the `ble_data` flash memory section, at the offset 500, and it checks its CRC32. If the process succeeds in the first 10 attempts, the challenge proceeds into checking the first 3 bytes of the string read, that should match `"Pri"`, checks that the byte in position 0x11 is higher than `'0'` and reads an integer from offset 0x18 using `atoi()`. It then uses the extracted integer as a key in a simple encryption function of a fixed plaintext. If the resulting ciphertext matches the expected one, the challenge succeeds. We can easily bruteforce in a few seconds the 32-bit integer which we have to use as a key using a quick C implementation of the encryption routine. The key appears to be 575703170.

The problem is now how to write the flash memory of the ESP32. Luckily the challenge does not reset that memory at boot, so writing the memory before flashing this challenge seems to be a valid solution. To write this memory we could simply write our own program, but let's first reverse engineer the `.heks` file.

The `.heks` file, seems to be a corrupted HEX file. An HEX file is an hexadecimal representation of the stripped data that must be loaded into the board memory. Each line of the file represents either memory content or some metadata, and ends with a checksum. Analysing the file we can immediately spot that it is invalid due to some CRC32 values being wrong. Using a small python script we can fix them and load the file correctly in Ghidra.

The file seems to be a challenge-like file, that instead of reading the ESP32 flash memory, writes it with a fixed string: `"Privilege Level: 0 Key: 29019203 1B3t9 "`. This is particularly handy, since the string is really similar to the one we need. It is sufficient to change the hexadecimal representation of the string in the `.heks` file, to the hexadecimal representation of our target string (`"Privilege Level: 1 Key: 575703170 1B3t9 "`), while fixing the CRC32 of the lines we edited in the file. Executing the file on the board, writes the string into flash memory. Therefore, simply executing the original challenge after having written the memory passes the checks and shows: `"f149=1 h4v3 bin REst0red!! Correct. Challenge Solved!!You won 200 points!! "`.

D - IMPACT

The challenge asks the user to find a collision on a custom hash function. It initializes a random number generator using NTP time information (which is predictable) and the CPU tick count (which is harder to predict). Then it generates an integer, prints it, and asks the user for a integer as well.

It checks that the provided integer is different from the generated one, and applies a custom hash function to the string representation of both integers. If the hash match in the lower 3 bytes, the challenge succeeds.

It is fairly easy to export the hash function to a C file, where we can bruteforce a 32 bit integer that collides on the last 3 bytes in a few seconds during the execution. Sending the right integer at runtime prints: `Flag:Y^-Wf'>H-KW2x<N. Challenge Solved!! You won 150 points!!.`

D - MOONLANDING

This challenge was one of the most complicated in this year's edition. The challenge implements a state machine which does different actions depending on the state. The machine starts at **state 0**, and behaves as follows. Each state once executed leads to the successive one, and all states are executed in a loop.

state 0) Opens the ESP32 WiFi access points, with an optional WPA_WPA2_PSK password. If the password is

provided the usual Access Point name gets changed to `ESC-2020-Secret` and the connection number limit is removed. The challenge starts with an empty password, so at first we will be able to connect to that. It additionally sets up the TCP server as usual.

state 1) Queries the SNTP Time of the ESP32 board, prints it if the debug mode is enabled, and saves it into a global buffer.

state 2) Checks the time string saved in the global buffer, encoding the first 16 bytes of that using a custom function, and comparing that to an expected result. If the check passes the challenges exits the state loop.

state 3) Waits for a connection to the listening server, and initializes the password buffer with a password depending on the exact tick on which the connection was received. Therefore the next time a WiFi access point will be created, it will have this password.

state 4) Reads a user input from the TCP connection, and performs checks on that. Sending a 256 byte long string, simply makes the challenge advance the state.

state 5) Resets the ESP32 module and waits 3 seconds.

state 6) Opens the ESP32 WiFi access point, using the same function as `state 0`. This time, the AP will be protected by password, so we will not able to connect to that unless we bypass it.

state 7) Enables debug prints, reads a string from the serial port and sends it as a command to the ESP32 module. It repeats this step until the capitalized string matches `"DEBUG MODE OFF"`. Once it matches, it disables debug prints, and goes into `state 1`.

Once the challenge exits the state loop, it waits for a connection to the TCP server, then checks the exit state from the loop, if it is a success state, it prints the flag.

The only way to exit the state loop cleanly, is through `state 2`. But to do that we must have the correct reply from the SNTP query. Since the board is not connected to the internet, and it has no way to know the real time, it is unclear what will be the result of the query. To understand that, we patched the binary itself to skip the check on debug mode, and always print the result of the query. Binary patching is especially useful when a debugging primitive is not quickly available to do small modifications to the executable. In this case it was sufficient to nop out the check on the debug global variable in the `degub_print()` function.

What turned out, is that the board receives a string like `"Thu Jan 1 00:00:18 1970"` as output from the query, probably starting counting the time from the UNIX null timestamp. We then bruteforced all the weekday, month, day, hour and minute combination to understand which was the one that would produce the desired hash using a python script, and obtained `"Wed Dec 31 20:09"` therefore it seemed there is no way to wait so long to connect to produce the right timestamp.

The fact that the board accepts command to forward to the ESP32 comes to a rescue for us. Indeed we can setup a fake NTP server that we will spawn in the same subnet to always reply a timestamp that matches the requested one. One such timestamp is `"Wed Dec 31 20:09:00 2014"`. Then if we send a command like `AT+CIPSNTPCFG=1,0,"<FAKE_NTP_SERVER_IP>"` the ESP32 will query the time to it, and sync its clock. Then if the SNTP query happens in at most a minute, the hash of the time will match.

The problem is that when the board will accept such serial commands, the only available WiFi network that the board exposes is the one protected by password.

We relied on binary patching as well to call different functions to print the tick state when the password was setup, to understand if the password was indeed predictable or bruteforceable. While the tick was not predictable, we were able to restrict the number of possible passwords to a range of about 200K different ones, due to the predictable way it is constructed.

Such an amount of password would be easily bruteforceable with a tool like hashcat [12], having an handshake dump [13], or a PMKID packet [14]. However, we did not have any of the required packets, since no one was connected to the board from which to obtain an handshake packet, and it seemed to not support PMKID packets. The amount of probable password is quite low so, it could be still possible to bruteforce that directly on the Access Point, as no connection limit exists, but it was an unreliable solution.

Indeed, we leveraged the fact that we were able to send commands directly to the ESP32 to reset ourselves the WiFi access point and create an unprotected one, which allowed us to connect to it, before sending the NTP configuration command.

Therefore the final solution to the challenge is:

1) Connect to the ESC-2020 Access Point and to the serial port of the board using `screen /dev/ttyACM0 115200`
2) Spawn a fake NTP server which will listen on 0.0.0.0:123 and always reply a timestamp corresponding to `"Wed Dec 31 20:09:00 2014"`
3) Connect to the TCP port of the challenge and send a 256 byte long string, like `"A"*256`. The ESP32 will then reset and spawn the new ESC-2020-Secret network
4) Wait for the challenge to accept commands on the serial port, and send the following command to reset the access point to one with no password. Quite often the commands will not succeed to be received from the ESP32, so it might be required to send a command multiple times, waiting a second between them, either by itself or by sending `AT` a few times in between to try and reset the command queue.

```
AT+CWMODE=0
AT+CWMODE=2
AT+CWSAP="ESC-TRX","",1,0,1,0
AT+CIFSR
AT+CIPAP="192.168.4.7"
AT+CIPSERVER=0
AT+CIPMUX=1
AT+CIPSERVER=1,50200
```

5) Now connect to the ESC-TRX access point which has no password, so that the board can connect to the SNTP server
6) Make the board sync with our custom NTP server, sending over serial the following command, which also ends the debug mode:

```
AT+CIPSNTPCFG=1,0,"<FAKE\_NTP\_SERVER\_IP>"
DEBUG MODE OFF
```

7) The board will echo back `DEBUG MODE OFF` over serial, and proceed to the NTP check. If the last two commands were sent with less than a minute passing beetween the two, the check will succeed.
8) Connect to the TCP port to receive the flag. Once connected we received:

```
Fl4g=oThisDaFl4gGYFl4gs
Challenge Solved!!You won 250 points!!
```

We automatized the solution with a python script, however we found the serial communication part being particularly unreliable, as the ESP32 board striving to always receive all the commands correctly.

D - PURSUIT

The challenge prints a base64 encoded string, which decoded is "`can you find me?`", and then asks for an input string. It then produces a 16 byte hash of the provided string, and checks with an expected one, if the hash matches the challenge is cleared.

The hash function is similar to the one we have seen in the challenge `hue` of setC, with a `SHA2` function, dividing the input into 4 chunks, applying a `SHA1` custom hash function on each of them, and some modular exponentiation operations. This time, the modular exponentiation operations combine the different results of the SHA1 function with the results of the previous modular exponentiation operations, so it seems much harder to bruteforce.

However, we can focus on the hint the base64 string gives us, and start searching for something odd in the binary. Looking at the `challenge_server()` function we can identify a strange static variable which was not present in the other challenges: `char challenge_server ::lexical_block_0 ::egg_1[5]`, which is initialized to the value "`http`". Using the readelf utility we can immediately find other similar variables: `forever_loop:: lexical_block_0:: lexical_block_0_0:: egg_2, power:: egg_3, SHA1:: egg_4, process_IPD:: egg_5, esp32_recvwait:: egg_6, LED_on:: egg_7`. Concatenating all the initialization values of these variables, we get the string `https://pastebin.com/wCdPGYah`. Accessing the link we see: `How did you find me?! The answer you are looking for is "345732_399_hun7".` Inserting the string the link suggests to us we receive from the challenge: `Correct Challenge Solved!!You won 100 points!!`

VIII. DISCUSSION

In this paper we presented how we approached the challenges of the ESC 2020 finals, and how we reasoned to solve them all. We showed how we were able to distributed the workload of solving the challenges between geographically distinct teammates using an emulator, which allowed to quickly prototype the solutions, before being able to test them on the real hardware.

The improvement we did on the emulator allowed us to cleanly hook the interactions – both with the environment and other modules like the ESP32 – to emulate the execution of the challenges which required a real hardware.

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 and improvement of our frontend for the Ghidra P-Code emulator to easily deal with such challenges. Not only our knowledge in the field increased substantially by solving those challenges, but we also improved a reusable tool for future tasks.

To facilitate analyses and extensions of our tool, the source code of improvements to the plugin will be made available after the end of the competition at:
https://github.com/TheRomanXpl0it/ghidra-emu-fun

Attachment I summarizes the input/output of the challenges.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] "CSAW Embedded Security Challenge." [Online]. Available: https://www.csaw.io/esc
[2] NSA, "Ghidra Software Reverse Engineering Framework." [Online]. Available: https://www.ghidra-sre.org/
[3] ——, "P-Code Reference Manual." [Online]. Available: https://ghidra.re/courses/languages/html/pcoderef.html
[4] Q. M. Chen and M. Almanza, "Ghidra Emulates Functions," 2019. [Online]. Available: https://github.com/TheRomanXpl0it/ghidra-emu-fun
[5] "Team TRX: CSAW 19 ESC Report." [Online]. Available: https://theromanxpl0it.github.io/articles/2019/11/11/csaw-esc-19.html
[6] "hal-fuzz: An HLE-based fuzzer for blob firmware ." [Online]. Available: https://github.com/ucsb-seclab/hal-fuzz
[7] "Unicorn — The Ultimate CPU emulator ." [Online]. Available: https://www.unicorn-engine.org/
[8] "Dragon Dance." [Online]. Available: https://github.com/0fffffffh/dragondance
[9] "Hifive1 rev b." [Online]. Available: https://www.sifive.com/boards/hifive1-rev-b
[10] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.
[11] "Columnar transposition cipher." [Online]. Available: https://en.wikipedia.org/wiki/Transposition_cipher#Columnar_transposition
[12] "hashcat." [Online]. Available: https://hashcat.net/hashcat/
[13] "Brute Forcing WPA/WPA2." [Online]. Available: https://www.shellvoide.com/wifi/hashcat-guide-how-to-brute-force-crack-wpa-wpa2/
[14] "How to crack wpa/wpa2 passphrase with pmkid." [Online]. Available: https://www.shellvoide.com/wifi/how-to-crack-wpa2-password-without-handshake-newly-discovered-method/

Attachment I: Summarization of all the input and outputs for the challenges. Please notice some of the I/O are not constant

Challenge	Input	Output
A - amnesia	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA [manual reset] AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA	+SYSFLASH:64,<AAAAAAAAAAAA*AAAAAAAAAAAA0200.*AAAAAAAAAAAAAAAAAAAAAAAAAAAAh9c Correct! Save key for report Correct!! You won 130 points!!
A - breakfast	4 ItsNotBaconian	4 ItsNotBaconian Correct!! You won 100 points!!
A - burst	3880128 2212 3545728	3880128 2212 3545728 Correct!! You won 50 points!!
A - flood	!!!! then, 4 times ”A”*0x400	*SaveforReport?TarbelaItaipu?Hoover?AtatuRK Correct!! You won 100 points!!
A - parthenon	youmeantotellmethatmyclassicalcipherisntsecure	Input: Ctxt: etemsletryaetyscrsemomtaahnuonlhciiieutlalcpsc youmeantotellmethatmyclassicalcipherisntsecure Correct!! You won 70 points!!
B - chase	[scripted interaction]	[...] Congrats you won Key:1*4mH3r3 Challenge Solved!!You won 100 points!!
B - esrom	remorseful	Correct Challenge Solved!!You won 70 points!!
B - middleman (original)	bytes.fromhex(’32323032203131363820313 2abf2b063’)	Correct. Challenge Solved!!You won 150 points!!
B - middleman (updated)	1341 1979 125 97	Correct. Challenge Solved!!You won 150 points!!
B - quiz	[scripted interaction]	Correct: 24233C+50152A- Challenge Solved!!You won 130 points!!
B - sequence	0 38	Correct. Challenge Solved!!You won 50 points!!
C - game	ENTER [wait for the correct time] ENTER	Correct!! Flag:D0u8l3d0x4+Wice;e2 1191 271182 [ChallengeSolved!! You won 150 points!!]
C - hue	THINKCOLORFULLY	Correct!! Challenge Solved!!You won 150 points!!
C - recycle	[scripted interaction]	Flag:Uoˆ/’B<a4100511e625e15562246032139314 1386e4c265a1c0f0c75 Challenge Solved!!You won 100 points!!
C - virtual1	ADD 2 2 1	Correct. Challenge Solved!!You won 50 points!!
C - virtual2	0 0 2 3 0 0 0 3 0 1 0 1 6 2 0 0 7 1 2 0 8 0 0 0	Correct. Challenge Solved!!You won 100 points!!
D - corrupt	[manual interaction to flash corrupt_solution.heks and then corrupt.hex]	f149=1 h4v3 bin REst0red!! Correct. Challenge Solved!!You won 200 points!!
D - impact	[scripted interaction]	Flag:Yˆ-Wf’>H-KW2x<N. Challenge Solved!!You won 150 points!!
D - moonlanding	[scripted + manual interaction]	Fl4g=oThisDaFl4gGYFl4gs Challenge Solved!!You won 250 points!!
D - pursuit	345732_399_hun7	Correct Challenge Solved!!You won 100 points!!