Kernel Sanders: CSAW ESC'20 Quals

Claire Seiler*, Hunter Searle*, Blas Kojusner*, Gabriella Neris*, Joseph Wilson*
*University of Florida, Gainesville, FL, USA
{ cseiler, huntersearle,bkojusner, gabriellaneris, jnw}@ufl.edu

I. Introduction

"The IoT integrates the interconnectedness of human culture – our 'things' – with the interconnectedness of our digital information system – 'the internet.' That's the IoT," said Kevin Ashton, the coiner of the term "Internet of Things." The promise of IoT's ability to bridge human social dynamics and culture and technology helped to popularize its adoption in the 2000s. Shortly after, smart devices could be found everywhere: in the home, in the car, in the workplace. The Internet of Things had veritably exploded.

However, with the ubiquity of IoT devices and the speed in which they were popularized, little emphasis was placed on security. Several high-profile attacks in which IoT devices were used as an attack vector brought attention to this critical issue. Default passwords, weak security measures, lack of updates, lack of an industry security standard, and weak/no encryption are among many of the security issues that continue to plague these devices.

The CSAE ESC 2020 competition aims to bring attention to the need for increased security in IoT devices. Through the exploitation of the firmware of a custom IoT WiFi access point, we aim to show how attackers can compromise and leverage these devices for malicious ends.

II. Challenge

To begin our analysis of the given `qual-esc2020.elf` object, we start by running the GNU `file` command on it.

```
qual-esc2020.elf: ELF 32-bit LSB
    executable, UCB RISC-V, version
    1 (SYSV), statically linked, not
    stripped
```

Immediately we know that this is a 32-Bit RISC-V ELF binary executable, which is not stripped, meaning functions should have names. Next running `strings` on the binary ("..." means snipped text) we see:

```
...
Correct, you solved Challenge 1!
Correct, you solved Challenge 2!
...
Correct, you solved Challenge 3!
Enter challenge number (1-3 or 4 to
    quit):
Enter exactly %d characters:
...
```

From the strings, we see an option to select 1-3 for a challenge, indicating there may be 3 challenges that we should look into.

With this initial analysis out of the way, we can try executing the binary to conduct some dynamic analysis. We made use of the helpful hints provided in the ESC 2020 repository to be able to do this.

```
$ chmod +x qual-esc2020.elf
$ qemu-system-riscv32 -nographic
    -machine sifive_e -kernel
    qual-esc2020.elf
```

Running the binary greets us with the prompt we saw when we ran the strings command. Each challenge produced the prompt requesting for a specific amount of characters. There was not much else we could do other than bruteforce input so we had to investigate further by using GHIDRA 9.0. We first added the riscv ghidra processor module, provided by the ESC repo, in our copy of Ghidra. We then started the program, created a new GHIDRA project and loaded the binary into it. We open the CodeBrowser tool and started with the auto-analysis. There are three challenges to be solved in the qualification binary. To do so, we implemented a variety of static and dynamic analysis reverse engineering techniques. For static analysis, we primarily utilized Ghidra for its disassembly and decompilation features. The RISC-V processor module provided to us by the challenge organizers assisted this process. We supplemented our analysis with command-line tools like strings. Dynamic analysis was conducted entirely using QEMU's RISC-V emulation to run the binary.

For the first challenge, we first examined the main function. The main function in this binary simply printed out the menu asking for the number of the challenge the user wanted to try out. Once a number is selected, the main function calls `challenge_x` where "x" is the number input. Ghidra outputs the following for `challenge_1`.

```
undefined4 challenge_1(char *param_1)

{
  undefined4 uVar1;
  char *local_24;
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  int local_14;

  local_20 = 0x2d435345;
  local_1c = 0x30323032;
  local_18 = 4;
  local_24 = param_1;
  if (((*param_1 == 'C') && (param_1[1]
      == 'S')) && (param_1[2] == 'E')) {
  local_14 = 3;
  while (local_14 < 0xb) {
    if (local_14 + -3 +
        (uint)(byte)param_1[local_14] !=
       (uint)*(byte *)((int)&local_24 +
           local_14 + 1)) {
    puts("Incorrect!");
    return 0xfffffffe;
    }
    local_14 = local_14 + 1;
  }
  puts("Correct, you solved Challenge
      1!");
  uVar1 = 0;
  }
  else {
  puts("Incorrect!");
  uVar1 = 0xffffffff;
  }
  return uVar1;
}
```

The first thing we noticed about this challenge is that it pushed several bytes onto the stack. Decoding these as ASCII text revealed a secret phrase: "ESC-2020". The function then has a simple check to ensure that the first three characters of the input is "CSE". After that, it loops over the remaining 8 character of the input. Each character has the value (i-3) added to it, where i is the loop index, then it is compared to a value stored on the stack, with an offset of (i+3). Checking the stack, we can see that the values it is compared against are the secret phrase. Thus it is a simple task of subtracting (i-3) from each character in the phrase to arrive at the correct input, using the following python code:

```
password = ESC-2020
i = 3
for char in password:
  print(chr(ord(char) + (i-3)))
  i += 1
```

Combining the output of the above code with "CSE" gives the input `CSEERA*.+,)`.

The Challenge 2 decompilation from Ghidra is as follows:

```
undefined4 challenge_2(int param_1)

{
  undefined4 local_24;
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  uint local_14;

  local_24 = 0x7a707a65;
  local_20 = 0x6f6d656c;
  local_1c = 0x7571736e;
  local_18 = 0x797a6565;
  local_14 = 0;
  while( true ) {
  if (0xf < local_14) {
    puts("Correct, you solved Challenge
        2!");
    return 0;
  }
  if ((*(byte *)((int)&local_24 +
      local_14) ^ 0x2a) != *(byte
      *)(local_14 + param_1)) break;
  local_14 = local_14 + 1;
  }
  puts("Incorrect!");
  return 0xffffffff;
}
```

From the decompilation, it can be seen that the function contains a for loop that iterates over 16 characters of a byte array residing at 65 7a 70 7a, 6c 65 6d 6f, and 6e 73 71 75 in memory (RISC-V is little-endian, so the hex addresses in the decompilation must be "reordered"). This byte array corresponds to the ASCII phrase 'ezpzlemonsqueezy'. Each character in the byte array is then xor-ed with 0x2a and compared with the current character in the input we passed in to the function. If all characters are equal, we get the solve message.

To retrieve the flag, we do a simple xor of each char in 'ezpzlemonsqueezy' with 0x2a. This produces `OPZPFOGEDY[_OOPS` which is the valid resulting key.

Challenge 3 seemed similar to the previous 2, with a for loop iterating over each character of the input, altering it, then comparing it to a set value. However, we found that the comparison phrase was first sent to another function arx(). This function took as inputs the secret phrase and the length of the input. It then went through several rounds of transformations detailed below.

1) initial code is ezpzlemonsqueezy
2) Do the following ten times
   - add 0x11 to each byte
   - shift each char to the left(with wraparound), and with 0xff
   - xor each char with 0xd
3) Take unsigned mod of each char by 0x3c, and with 0xff
4) Add 0x22 to each char

This process can be visualized in the challenge code below.

```
undefined4 challenge_3(int param_1)

{
  undefined4 local_24;
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  uint local_14;

  local_24 = 0x7a707a65;
  local_20 = 0x6f6d656c;
  local_1c = 0x7571736e;
  local_18 = 0x797a6565;
  arx(&local_24,0x10);
```

```
    local_14 = 0;
    while( true ) {
    if (0xf < local_14) {
      puts("Correct, you solved Challenge
          3!");
      return 0;
    }
    if (*(char *)((int)&local_18 + (3 -
        local_14)) != *(char *)(local_14 +
        param_1)) break;
    local_14 = local_14 + 1;
    }
    puts("Incorrect!");
    return 1;
  }
```

After these transformations, the input is compared, in reverse order, to the phrase returned by arx(). To solve the challenge, we simply wrote a program that ran all of the steps, then reversed the output to get the correct input: I0E;3.<2<3G<33C#

```
#include "stdio.h"

int main(){
int code[0x10] =
    {0x65,0x7a,0x70,0x7a,0x6c,0x65,0x6d,0x6f,0x6e,0x73,0x71,0x75,0x65,0x65,0x7a,0x79};
for(int i = 10; i > 0; i--){
  for (int a = 0; a < 0x10; a++){
    code[a] = (code[a] + '\x11') & 0xff;
  }
  int first = code[0];
  for(int b = 0; b < 0xf; b++){
    code[b] = code[b+1];
  }
  code[0xf] = first;
  for(int c = 0; c < 0x10; c++){
    code[c] = code[c] ^ 0xd;
  }

}
for(int d = 0; d < 0x10; d++){
  code[d] = ((unsigned)code[d] % 0x3c) &
      0xff;
  code[d] = code[d] + 0x22;
}
for(int out = 0xf; out >= 0; out--){
  printf("%c", code[out]);
}
printf("\n");
}
```

Overall, our reverse engineering techniques were successful in solving these three challenges. However, for solving more complex RISC-V challenges, we will most likely not default to manual static analysis. We aim to leverage symbolic execution and symbolic analysis frameworks to assist us with solving more complex binaries. Angr, a python framework developed by UC Santa Barbara, is well-suited for such a task. Using angr's RISC-V lifter, we should be able to

III. Conclusion

To complete this qualifier challenge, our team used GHIDRA, a reverse engineering tool, along with a riscv ghidra processor module to work backwards through an unfamiliar binary file to obtain the correct flag. For the first challenge, we collected the flag `CSEERA*.+,)`. For the second, the flag `OPZPFOGEDY[_OOPS` was quickly obtained. For the final challenge, we built off of knowledge gained from the previous challenge and produced the flag I0E;3.<2<3G<33C#. From the experiences gained through this qualifier, we feel well-adjusted to proceed with the firmware-level analysis and exploitation of different RISC-V programs.