Kernel Sanders: CSAW ESC'19 Quals

Grant Hernandez*, Claire Seiler*, Owen Flannagan*, Hunter Searle*, Kevin R.B. Butler*
*University of Florida, Gainesville, FL, USA
{grant.hernandez, cseiler, owenflannagan, huntersearle, butler}@ufl.edu

I. Introduction

The advent of Radio Frequency Identification (RFID) technology has quickly been followed by its rapid and pervasive integration across multiple industries, from transportation to security. However, with its widespread adoption comes an increased need to investigate the potential security implications of integrating RFID technology. As RFID is commonly utilized for a wide variety of authentication, access control, asset tracking, payment, and identification applications, these systems could be vulnerable to attacks that exploit the underlying RFID technology.

RFID systems typically consist of three main components: an RFID tag, an RFID reader, and an antenna. A reader, which is a two-way radio transmitter-receiver, sends radio frequency signals to tags and reads the response. Tags, which store data like serial numbers, can be read-only or read/write. Additionally, the tags may also be designated as passive or active, meaning they are powered by the radio energy transmitted by the reader or by an on-board battery, respectively.

These systems raise a variety of security concerns; they are potentially vulnerable to a variety of eavesdropping, spoofing, or jamming methods. Additionally, common reverse engineering and firmware exploitation techniques can be applied to the exploitation of RFID readers.

RFID readers, like other embedded devices, contain firmware in non-volatile, flash memory. This firmware can be extracted from a physical memory chip using tools like *flashrom*, *binwalk*, Bus Pirates, and logic analyzers. Alternatively, if available, JTAG/SWD could be used to perform a memory dump of the running CPU if accessing the firmware via NOR/NAND flash is too difficult. After extraction, static and dynamic analysis can be done to identify potential vulnerabilities that could lead to compromise of the reader. In static analysis, disassemblers like GHIDRA, IDA Pro, radare2, or Binary Ninja can be leveraged to analyze the assembly instructions corresponding to the firmware. A decompiler can assist in understanding and to recreate the source code in a high level language.

Once disassembled or decompiled, specific vulnerabilities can be identified and targeted exploits can be developed. Vulnerabilities like stack- and heap-based buffer overflows, off-by-one errors, integer overflows, uncontrolled format strings, poor input validation and sanitization, OS command injection, disabled (but not removed) debugging functionality, and hardcoded credentials often plague embedded firmware, which typically rely on languages with manual memory management like C or C++. Thus, these types of vulnerabilities can serve as a guide to analyzing the disassembled firmware of an RFID reader.

After identifying a vulnerability, a targeted exploit can be created. Mitigations like stack canaries, heap protection, ARM's specific eXecute Never (XN), RELocation Read-Only (RELRO), Position-Independent Executable (PIE), and Address Space Layout Randomization (ASLR) can be overcome with some ingenuity and techniques like return-oriented programming (ROP) chaining, stack smashing, heap spraying, information disclosures and more. Exploit writing frameworks, such as pwntools, can assist in developing an exploit for an RFID reader's firmware, depending on the device architecture and the protections enabled.

II. Challenge

To begin our analysis of the given `qualification.out` object, we start by running the GNU `file` command on it.

```
qualification.out: ELF 64-bit LSB
    executable, x86-64, ... , not
    stripped
```

Immediately we know that this is an x86-64 ELF binary executable, which is unstripped, meaning functions should have names. Next running `strings` on the binary ("..." means snipped text) we see:

```
...
Great Job! The flag is what you entered
The flag is <<shhimhiding>>
;*3$"
GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.4)
    4.8.4
...
qualification.cpp
...
_Z14secretFunctionv
...
_Z17challengeFunctionPc
```

From the strings, we see a "good flag" message, an actual flag, that this binary was written as C++, and two C++ mangled functions.

With initial static analysis out of the way, we can set the file as executable and do some dynamic analysis.

```
$ chmod +x qualification.out
$ ./qualification.out
$ ./qualification.out test
```

```
$ ./qualification.out shhimhiding
```

Running the binary with and without arguments (even the flag found via strings) yields no “goodboy” message. To investigate further, we start GHIDRA 9.0 to begin our analysis. We create a new GHIDRA project and load the binary into it. We open the CodeBrowser tool and perform auto-analysis. The first step in solving this challenge was to look at the main function. This is a simple function that checks if exactly 2 arguments were passed to the program, then calls `challengeFunction` that takes a char* as it’s only parameter. Ghidra outputs the following for `challengeFunction`.

```
void challengeFunction(char *param_1)
{
  bool bVar1;
  int local_2c;
  uint local_28 [4];
  undefined4 local_18;
  undefined4 local_14;
  undefined4 local_10;
  undefined4 local_c;

  local_28[0] = 1;
  local_28[1] = 2;
  local_28[2] = 1;
  local_28[3] = 2;
  local_18 = 1;
  local_14 = 2;
  local_10 = 1;
  local_c = 2;
  bVar1 = true;
  local_2c = 0;
  while (local_2c < 8) {
    if (((int)param_1[(long)local_2c] -
        0x30U ^ 3) !=
        local_28[(long)local_2c]) {
      bVar1 = false;
    }
    local_2c = local_2c + 1;
  }
  if (bVar1) {
    puts("Great Job! The flag is what
        you entered");
  }
  return;
}
```

After all the definitions and initialization, the important part of this function is in the while loop. The loop iterates through each of the first 8 chars of the input, applies a simple transformation, then compares it to the corresponding indices of the array, local_28. If each comparison is true, the function prints out a success message. Otherwise, it exits. In order to figure out what input was required, we worked backwards from the local variable. The first 4 numbers in the array are 1, 2, 1, and 2, which are explicitly assigned to the first 4 indices of local_28. Because the array is only allocated with a size of 4, the last 4 comparisons in the while loop run off the end of the array. Space for local variables is allocated on the stack, so the 4 memory spaces immediately after local_28 are the next 4 local variables allocated, namely local_18, local_14, local_10, and local_c, with values 1, 2, 1, and 2, respectively. So, after applying the transformation on the input, the first 8 chars must be equal to 1, 2, 1, 2, 1, 2, 1, and 2. The last step is to reverse the transformation, which consists of subtracting the hex value 30, the XORing with 3. The XOR operation turns a 1 into a 2, and a 2 into a 1. Adding 0x30 gives the numerical value of our input as 0x32, 0x31, 0x32, 0x31, 0x32, 0x31, 0x32, and 0x31. Consulting an ASCII table gives the char value for this sequence as "21212121". Running the program with that argument prints out the success message.

Based off of our reverse engineering, we can rename variables and change types to the following:

```
void challengeFunction(char *flag) {
  int i;
  uint table [8];
  bool goodFlag;

  table[0] = 1;
  table[1] = 2;
  table[2] = 1;
  table[3] = 2;
  table[4] = 1;
  table[5] = 2;
  table[6] = 1;
  table[7] = 2;
  goodFlag = true;
  i = 0;

  while (i < 8) {
    if (((int)flag[(long)i] - 0x30U ^ 3)
        != table[(long)i]) {
      goodFlag = false;
    }
    i += 1;
  }
  if (goodFlag) {
    puts("Great Job! The flag is what
        you entered");
  }
  return;
}
```

A. A deeper look at the assembly

In order to understand how to reach the code that puts the affirmative message, it is important to understand how to prevent goodFlag from being set to False. As goodFlag is initialized to true, it is necessary to avoid the conditional passing. To better understand this code, we looked at this region as x86 assembly.

```
LAB_0040057e            XREF[1]:
    0x4005b6(j)
0x40057e 8b45dc     MOV    EAX, dword ptr
     [RBP + local_2c]
0x400581 4863d0     MOVSXD RDX, EAX
0x400584 488b45c8   MOV    RAX, qword ptr
     [RBP + local_40]
0x400588 4801d0     ADD    RAX, RDX
0x40058b 0fb600     MOVZX  EAX, byte ptr
     [RAX]
```

```
0x40058e 8845db     MOV     byte ptr [RBP + local_2d], AL
0x400591 0fbe45db   MOVSX   EAX, byte ptr [RBP + local_2d]
0x400595 83e830     SUB     EAX, 0x30
0x400598 83f003     XOR     EAX, 0x3
0x40059b 89c2       MOV     EDX, EAX
0x40059d 8b45dc     MOV     EAX, dword ptr [RBP + local_2c]
0x4005a0 4898       CDQE
0x4005a2 8b4485e0   MOV     EAX, dword ptr [RBP + RAX*0x4 + -0x20]
0x4005a6 39c2       CMP     EDX, EAX
0x4005a8 7404       JZ      LAB_004005ae
0x4005aa c645da00   MOV     byte ptr [RBP + local_2e], 0x0
```

In this assembly, `RBP + local_2c` holds the value of i that increments from 0 to 8. Additionally, `RBP + local_40` holds the parameter that is passed to this `challengeFunction`, and this is the argument to the program itself. When RAX and RDX are added at 0x400588, this is used to create a pointer to the i'th character of the string, and this character is moved into EAX and `RBP + local_2d`. After 0x30 is subtracted from this value, it is xored with 0x3. 0x30 is notable because this is the ascii value for the character '0', so subtracting 0x30 from any character of a one digit integer would retrieve it's value.

The incrementing value i is moved into EAX again at 0x4005a0, and this time it is multiplied by 0x4 and added to `RBP - 0x20`. This is where the array of 0's and 1's is stored, and this is statically created at the beginning of the function. When these are compared, execution will jump to 004005ae if they are equal, and 0 is moved into `RBP + local_2e` if not. This local variable holds the boolean that we need to remain 1. Luckily, xor is a reversible operation, and addition is as well. 1 xored with 3 is 2, and 2 + 0x30 is 0x32. This is the character '2' in ASCII. 2 xored with 3 is 1, and 1 + 0x30 is 0x31, or '1' in ASCII. Since we know the order in which the values of 1 and 2 are assigned into the static array, we can determine that the argument to give the program is `21212121`. Running `./qualification.out` with the argument of 21212121 gives the affirmative message.

Further investigation of the functions discovered by GHIDRA, we notice one named `secretFunction`.

```c
void secretFunction(void) {
  puts("The flag is <<shhimhiding>>");
  return;
}
```

This function is never referenced by the `main` or `challengeFunction`, but it was easily discovered through static analysis (GNU `strings` also revealed the other flag string.

III. Conclusion

In this qualifier we used GHIDRA to reverse engineer an unknown binary file to understand how to provide the correct flag and discover any other interesting features. We recovered the correct flag of "21212121" and noticed the false, hidden flag of "shhimhiding". With the necessary background in reverse engineering we are prepared to tackle the firmware analysis and exploitation of the RFID platform.