Embedded RFID Firmware Reverse Engineering

Abstract—**RFID systems are used in many critical devices in our world. They can be present along a range of frequencies in devices from access cards for secure facilities to NFC chips which help power modern credit cards. While these systems are widely used, most of the authentication logic is computed by devices which receive data/signals from RFID chips. Thus, to attack these systems it is paramount to reverse engineer the embedded firmware which powers RFID-enabled systems.**

Index Terms—**Binary exploitation, RFID, reverse engineering**

I. Introduction

As the world becomes more advanced, methods of authorization have evolved. Much like how in the mobile space access has often changed from being restricted by a pin to biometrics, many things like secure doors and credit cards are secured by RFID chips embedded into cards.

These are often deemed to be more secure because of a lower chance for end users to have credentials stolen or cards cloned. While users' RFID chips provide details for authentication, that authentication is often done by embedded firmware which uses the data read from RFID chips.

In this paper we focus on approaches to reverse engineer such firmware that allows embedded systems to decide if a user's RFID identifier is valid or not. We then employ these approaches on the provided CSAW ESC 2019 challenge binary, and report the results.

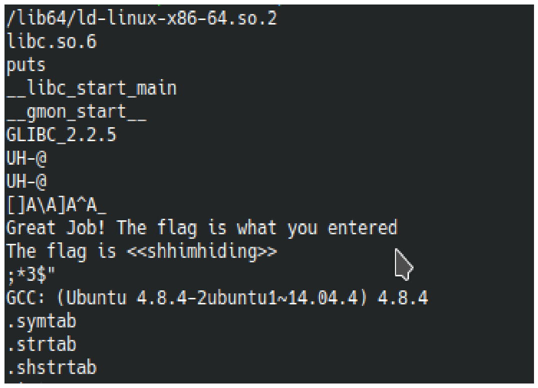
The rest of the paper is organized as follows. Section II discusses the background of the paper, defining terminology and concepts. Later, in Section III, solutions to the CSAW ESC 2019 qualifications binary are presented. Approaches and results used for Finals round challenges are presented in IV, V, and VI. Finally, we conclude our paper in Section VIII.

II. Background

In this section we present and define background concepts which are applicable to reverse engineering and exploiting RFID-based systems.

A. Static Analysis

Static analysis is the process of analysing a binary, firmware image, or piece of hardware in a off/static state. This is the opposite of dynamic analysis, which is analysing a binary while its running. For firmware, static analysis typically involves using a disassembly to produces assembly based on the compiled firmware machine instructions.



```
/lib64/ld-linux-x86-64.so.2
libc.so.6
puts
__libc_start_main
__gmon_start__
GLIBC_2.2.5
UH-@
UH-@
[]A\A]A^A_
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
.symtab
.strtab
.shstrtab
```

Fig. 1. Printable strings from challenge binary

B. Ghidra SRE

Ghidra Software Reverse Engineering [1] tool is a static analysis framework that provides analysis and views for a binary or firmware image. We will be using Ghidra to disassemble, and decompile the provided qualification binary and ultimately understand its functionality.

C. Firmware Exploitation

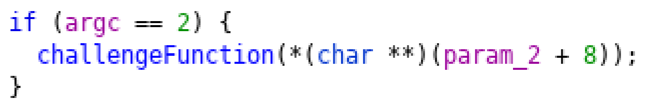
Undefined behavior in a system that can be harnessed by an attacker is known as vulnerability. The usage of a vulnerability to induce anomalous behavior is what is known commonly as an exploit. Firmware Exploitation is the act of exploiting vulnerabilities inside the firmware code, typically execute attacker-defined code, or bypass authentication. Stack based buffer overflows, use-after-frees, and return orient programming are common exploit techniques that can be used in firmware exploitation.

D. Symbolic Execution

During symbolic execution a program is analyzed at each stage. Symbolic values are used for variables at each stage, then upon reaching a desired state, a solver is used to solve those symbols for valued inputs which produce the desired results [7]. One popular symbolic execution tool used later in this work is angr [6].

E. Fuzzing

Fuzzing is a technique for discovering vulnerabilities in a system where test cases are created which generates and provides mutated input to a system and monitors for exceptions. Fuzzing embedded targets can be made more difficult because those embedded devices require customized harnesses for



```
if (argc == 2) {
  challengeFunction(*(char **)(param_2 + 8));
}
```

Fig. 2. Decompiled functionality used to trigger the challengeFunction

providing that mutated input to device programs. Thus, many embedded targets require custom written fuzzers. However there are many publicly available fuzzers such as AFL [3], and Radamsa [2] that work for general programs, and can be harnessed for embedded devices.

F. *RFID Security*

While RFID cards/chips can come in a variety of frequencies ranging from low frequency to ultra-high frequency, the devices tend to work in similar fashions. Historically these devices contain some string that is used for identification [9], whether it be a person's ID card or a product in a store [9].

Historically the largest threat to RFID based identification systems are spoofing and cloning of real identifications. Oftentimes this is done by being in close proximity of the target and using physical means for cloning IDs [8].

As the field has progressed, there have been pushes to make the systems more complex, utilizing different cryptographic schemes to either block or increase the cost of historical attacks [8] [9]. While these do help with protecting against attacks, there is still the notable attack vector of dumped firmware. Even though it does not render the system insecure by itself, if an attacker is able to dump the firmware of an RFID receiver/scanner, they would be able to begin to test the firmware for vulnerabilities or logical flaws in code (for example flaws in cryptography algorithms) that might allow them to mount an attack against the system.

III. Qualification Binary

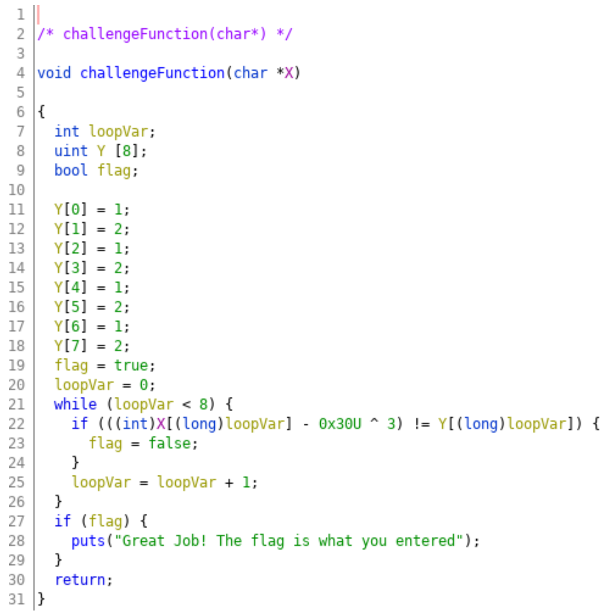
In this section we present approaches and solutions to the provided qualification binary. We start with a static analysis approach, then explore automated solutions, which could make solving complex binaries more feasible.

A. *Static Analysis*

1) Initial analysis and strings: The goals during initial analysis are to obtain high level understandings of the qualifications executable characteristics such as architecture, exploit mitigations, and reactionary behavior when supplied with various forms of input. That input could be in the form of stdin, arguments, or input files, as an example.

Using file, the challenge binary is shown to be a 64-bit x86 binary. This binary contains several print-able strings, which are revealed using the strings command line utility, and are shown in Figure 1. Many of these strings are functions, section headers, and other garbage data. However, checking the printable strings revealed two possible flags: "Great Job! The flag is what you entered" and "The flag is shhimhiding".

Knowing that the first flag is "shhimhiding" leaves the authors to find what triggers the former string.



```
 1 
 2 /* challengeFunction(char*) */
 3 
 4 void challengeFunction(char *X)
 5 
 6 {
 7   int loopVar;
 8   uint Y [8];
 9   bool flag;
10 
11   Y[0] = 1;
12   Y[1] = 2;
13   Y[2] = 1;
14   Y[3] = 2;
15   Y[4] = 1;
16   Y[5] = 2;
17   Y[6] = 1;
18   Y[7] = 2;
19   flag = true;
20   loopVar = 0;
21   while (loopVar < 8) {
22     if (((int)X[(long)loopVar] - 0x30U ^ 3) != Y[(long)loopVar]) {
23       flag = false;
24     }
25     loopVar = loopVar + 1;
26   }
27   if (flag) {
28     puts("Great Job! The flag is what you entered");
29   }
30   return;
31 }
```

Fig. 3. Decompiler output for challengeFunction

Another command line utility "checksec" was used to see if there were any exploit mitigations present in the binary. The only protection reported was Data Execution Prevention, which allows for regions of memory to be specified as non-executable.

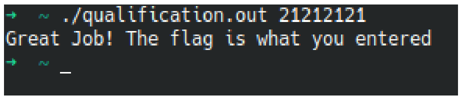
2) Reverse engineering and the algorithm: Using Ghidra to disassemble and decompile the executable, starting with the main function, it is shown that the program expects command line arguments to be given. The pointer to the first argument will then be passed over to the function "challengeFunction" as shown in Figure 2.

Decompiling the "challengeFunction" (Figure 3) will reveal to use the print statement that will give us the desired output of "Great Job! The flag is what you entered" along with some operations and "if statements" needed to be true to execute that print statement. The decompilation of the function is below with some retyping and changes to variable names to make cleaner.

With the decompilation above, the first assignment we see is integers of 1's and 2's being assigned to an integer array Y. Then, a while loop iterating though 8 characters of the command line argument. Operations are then performed on each byte, X, and eventually compared to the value in the integer array according to the index of the loop variable. If after all the operations occur, and all of the results match with the value in the array Y, according to index, we will have the "flag" variable left unchanged and the condition meet to print out the success string. The operations can be see on line 22 inside the if statement, and can be represented by the function below.

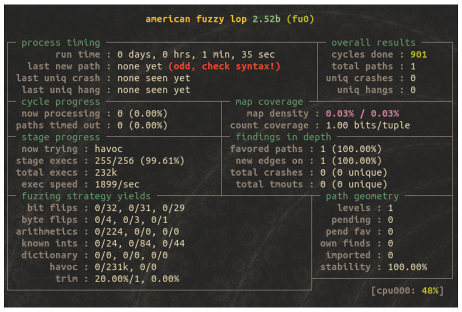
$$f(x) = (x - 0x30) \oplus 3 \tag{1}$$

3) Solution: With the information pulled from reverse engineering, its deduced that the program will take a single



```
→  ~ ./qualification.out 21212121
Great Job! The flag is what you entered
→  ~ _
```

Fig. 4. Output from program when entering the second flag



```
                 american fuzzy lop 2.52b (fu0)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 1 min, 35 sec       │  cycles done : 901 │
│   last new path : none yet (odd, check syntax!)      │  total paths : 1   │
│ last uniq crash : none seen yet                      │ uniq crashes : 0   │
│  last uniq hang : none seen yet                      │   uniq hangs : 0   │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────┤
│  now processing : 0 (0.00%)         │    map density : 0.03% / 0.03%    │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple  │
├─ stage progress ────────────────────┼─ findings in depth ───────────────┤
│  now trying : havoc                 │ favored paths : 1 (100.00%)       │
│ stage execs : 255/256 (99.61%)      │  new edges on : 1 (100.00%)       │
│ total execs : 232k                  │ total crashes : 0 (0 unique)      │
│  exec speed : 1899/sec              │  total tmouts : 0 (0 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────┤
│   bit flips : 0/32, 0/31, 0/29                      │    levels : 1      │
│  byte flips : 0/4, 0/3, 0/1                         │   pending : 0      │
│ arithmetics : 0/224, 0/0, 0/0                       │  pend fav : 0      │
│  known ints : 0/24, 0/84, 0/44                      │ own finds : 0      │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : 0      │
│       havoc : 0/231k, 0/0                           │ stability : 100.00%│
│        trim : 20.00%/1, 0.00%                       ├────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 48%]
```

Fig. 5. Attempted AFL fuzzing, note the single total path

command line argument, iterating through the the first 8 characters, and subtracting each of them by 0x30 and xor'ing the result by 0x03. The final result of that, X, will then be compared to an integer in the array [1, 2, 1, 2, 1, 2, 1, 2], Y, according to its index (i.e $X[4] == Y[4]$). Using

$$f(x) = (x - 0x30) \oplus 3 \tag{2}$$

and the ASCII table to find the literal character '2' and '1' hold the numeric values of 0x32 and 0x31, which when used in the equation will result in the numeric 1's and 2's in the array. Providing the string "21212121" as the binary's first argument will result in the correct match for the values in the integer array, and the program outputting "Great Job! The flag is what you entered".

Thus the two flags recovered from static analysis of the challenge binary are a command line argument of "21212121" and the string "¡¡shhimhiding¿¿" from a secret function inside of the binary.

B. Automated Solutions

While static analysis faired well for this challenge, more complex programs and algorithms might be solved more efficiently with automated solution. In this section we cover some approaches we took to attempt to solve the challenge binary, with varying success.

1) Symbolic Execution: For symbolic execution we used angr [6]. It is a python library which allows users to specify which features of a program to search for, then searches to execute the program in a way that satisfies those specifications.

In our testing we assumed that the researcher who was looking to reverse engineer the program was able to the basic information from static analysis which includes the argument calls which trigger challengeFunction.

From then we had angr find a suitable input that triggered an output containing the words 'flag' since we knew that we needed to trigger a print statement which included the word flag. The symbolic solution script then prints the input that led to that print. These files are attached to the hotcrp submission in .tar.gz form for artifact evaluation and future reproducability.

While running the solution script, assuming that the input was a string of integer values, it took only 16 seconds to reach the solution by itself. This time was measured on a Haswell processor manufactured in 2014. To simulate not knowing the keyspace of the input, the same solver was used with a keyspace of all printable bytes as input. That solution took 16.2 seconds, showcasing the strength of symbolic execution when working with larger keyspaces. While this may be relatively timely to develop and deploy when compared to looking at the decompiler output while doing static analysis, it is feasible that symbolic execution would help alleviate the necessary work for more complex algorithms, or larger programs.

IV. Finals Round Challenges

During the finals round fives sets of challenges were released, with varying levels of difficulty. The authors took two approaches, each with different difficulties and achievement. These included symbolic execution with Angr, and static analysis with Ghidra.

V. Angr in the Finals

As a result of most of the challenges comprising of inputs that satisfy conditions within the binary, it was thought that Angr would suit these nicely, similarly to with the qualification binary.

The first difficulty when using Angr to automatically analyze and attempt to solve challenges was that the Cortex-M processor on the TeensyDuino uses Thumb2 code. Prior to manually setting angr to use this architecture Vex would generate invalid instructions and fail to decode some functions at all.

After setting the correct architecture settings, Angr was able to traverse some challenge binary functions. The shortcoming with this approach was that it did not result in found paths, but rather each challenge that was traversed resulted in unconstrained variables, which are unsolvable by Angr. Thus, there were no finals round challenges solved using Angr.

VI. Static Analysis of Finals Challenges

Static analysis was successfully used to solve a few of the finals round challenges. While stairs from challenge set A was a simple XORing problem, other challenges proved to be a bit more complex to decipher. Those challenges that were solved, their set, and related solution hashes are presented in section VI.

VII. Solved Finals Challenge Hashes

In this section we present the challenges that the authors have solved, and the resulting challenge hashes.

challenge	set	hash
stairs	a	396f4b1cdf1cc2e7680f2a8716a18c887cd489e12232e75b6810e9d5e91426c7
recess	c	370815b8d8fde829f5c35f893d0b4139d61a775baa4181fcac1fffe014bde9ea
break	c	2d3448f09329f453e6f3a5403d89c061a9dabfbb9103ad6b8cc86d16345a7547

VIII. Conclusion

In this paper we have presented the basics of reverse engineering and exploiting the firmware that provides functionality for RFID-based systems. Both manual static analysis and automated approaches such as symbolic execution and fuzzing were presented. All approaches were then used to attempt to solve the CSAW ESC 2019 qualification binary, with varying results explained. Both flags ”21212121” and ”<<shhimhiding>>” were successfully recovered through those efforts. Static analysis was then used to solve multiple CSAW ESC finals round binaries.

References

[1] National Security Agency “Ghidra”. National Security Agency, 2019. https://ghidra-sre.org/

[2] Aki Helin “Radamsa, a general-purpose fuzzer”. gitlab.com, 2019. https://gitlab.com/akihe/radamsa

[3] Michal Zalewski “american fuzzy lop”. lcamtuf.coredump.cx, 2017. http://lcamtuf.coredump.cx/afl/

[4] Michal Zalewski “american fuzzy lop - sample argv fuzzing wrapper”. github.com, 2015. https://github.com/mirrorer/afl/blob/master/experimental/argv_fuzzing/argv-fuzz-inl.h

[5] Andrew Griffiths, Michal Zalewski “ american fuzzy lop - high-performance binary-only instrumentation”. github.com, 2017. https://github.com/mirrorer/afl/blob/master/qemu_mode/patches/afl-qemu-cpu-inl.h

[6] Shoshitaishvili, Yan and Wang, Ruoyu and Salls, Christopher and Stephens, Nick and Polino, Mario and Dutcher, Audrey and Grosen, John and Feng, Siji and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. IEEE Symposium on Security and Privacy, 2016.

[7] James C. King “Symbolic Execution and Program Testing”. Communications of the ACM, 1976.

[8] Government of Hong Kong Special Administrative Region “RFID SECURITY”. infosec.gov.hk, 2008. https://www.infosec.gov.hk/english/technical/files/rfid.pdf

[9] Francis Brown “RFID Hacking Live Free or RFID Hard”. DEF CON 21, 2013. https://www.defcon.org/images/defcon-21/dc-21-presentations/Brown/DEFCON-21-Brown-RFID-Hacking-Updated.pdf