CSAW 2019 Embedded Security Challenge Report

Adam Verner
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
vernead2@fit.cvut.cz

Jan Havranek
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
havraja6@fit.cvut.cz

***Abstract*—This document represents our approach to solving CSAW19 embedded security challenge qualification round.**

I. Introduction

In the first part we talk about analysis of unknown files and what we can learn from standard Linux programs like *file*, *strings*, *strace* and so on.

Next, we address the basic reverse engineering techniques of static and dynamic analysis, talk about reverse engineering the challenge executable and show parts of the decompiled program.

The last section discusses side-channel attacks and their mitigation.

II. File analysis

We are presented with single file called ”qualification.out”.

A. File identification

To identify the challenge file we used program *file* which is used to identify the type of data based on magic bytes matching and pattern recognition.

```
ELF 64-bit LSB executable,
x86-64,
version 1 (SYSV),
dynamically linked,
interpreter /lib64/l,
for GNU/Linux 2.6.24,
not stripped
```

Fig. 1. ”file” output

In figure 1 we can see the output, which tells us, that the file is x86-64 executable and is dynamically linked. This info will be useful when we will need to analyze the binary further using other software.

B. Searching for strings

Strings is command which looks for strings of printable characters inside the file.

Running strings with ”-d” only scans the data sections in the file reducing output noise. This scan yields some really interesting results.

```
/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$"
```

Fig. 2. strings output

On first 6 output lines in figure 2 we can see that the program was dynamically linked, glibc is used in the executable and that GCC was used to compile and link the executable.

Apart from that we can see the three really interesting strings. There are two strings telling us what the flag is. As there is no submission system for the flag we’ll have to inspect the program to see what it really does.

1) vector mitigation: There is no universal solution to this problem. If we want to keep people with malicious intention from read strings we could use some simple encoding and then decode the string before it’s printing. But that could still be reversed. If there are some secrets that should be viewed only by authorized user, they should be encrypted using user’s password and some strong algorithm, e.g. AES.

III. Dynamic analysis

Dynamic analysis approach is to runs the executable inside controlled environment and inspects it’s behaviour. Unfortunately none of the described techniques could be used on the provided binary as it doesn’t make any calls to system nor to any dynamically linked library that could be intercepted.

A. strace

Is a utility that runs provided executable and examines all calls to the kernel a prints them.

B. ltrace

ltrace is in behaviour similar to strace, it intercepts calls to dynamically linked libraries

C. ldd

ldd prints shared object dependencies which we can in attack phase intercept with our own libraries. For example we can intercept call to *strcmp* and always return true.

IV. Static code analysis

A. Tools

All of the aforementioned approaches are really useful, but they do not tell us what really happens under the hood.

To learn what the executable is doing, we have to dive into it's sources. The simplest tool to disassemble ELF executable is *objdump*. Reading the raw assembly takes a lot of time and requires a lot of experience. Using a specialized tool can save us a lot of work.

There are many similar tools available, but most of them are proprietary and require expensive licenses. The best known tools in this category are IDA or Binary ninja.

This year, at the begging of March, the National Security Agency (NSA) released their software reverse engineering framework called Ghidra[1] under an open source license. It is quite powerful and functionally comparable with the other mentioned expensive software. It performs detailed analysis of the provided binary and makes its reverse engineering a lot easier.

B. Analyzing the code

Static analysis generally starts with examining the program's entry point. As is the case with all C programs, we start from the *main* function shown in figure 3.

```
int main(int argc, char **argv) {
  if (argc == 2) {
    challengeFunction(argv[1]);
  }
  return 0;
}
```

Fig. 3. Decompiled *main* function

After minor edits of the function's signature in Ghidra, we can see right away that the program expects exactly one command line argument which is directly passed to *challengeFunction*.

The decompilation of *challengeFunction* was not perfect right away but Ghidra got quite close. Again, with minor improvements and refactoring, the function, shown in figure 4, is very well readable and quite easy to understand.

This function accepts the command line argument from *main* function and further processes it. It iterates over eight character of the argument. For each character it performs some

[1]https://ghidra-sre.org/

```
void challengeFunction(char *input) {
  int arr[8];
  arr[0] = 1;
  arr[1] = 2;
  arr[2] = 1;
  arr[3] = 2;
  arr[4] = 1;
  arr[5] = 2;
  arr[6] = 1;
  arr[7] = 2;

  bool correct = true;
  for (int i = 0; i < 8; i++) {
    if ((input[i] - 0x30 ^ 3) != arr[i]) {
      correct = false;
    }
  }

  if (correct) {
    puts("Great Job! The flag is"
         "what you entered");
  }
}
```

Fig. 4. Decompiled *challengeFunction*

computation and compares the result with a value in predefined local array, here called *arr*.

The mentioned computation is quite simple. Firstly, 0x30 is subtracted from the current character which is an operation that converts an ASCII representation of a digit to the actual value. For example, the character '5' has ASCII code 0x35 and after subtracting 0x30, we are left with the actual number five. Next, the result XORed with the number three. With this information, we can simply reverse the operations and find the password.

$$arr[0] \oplus 3 + 0\text{x}30 = 1 \oplus 3 + 0\text{x}30 = 0\text{x}32 = \texttt{'2'}$$

$$arr[1] \oplus 3 + 0\text{x}30 = 2 \oplus 3 + 0\text{x}30 = 0\text{x}31 = \texttt{'1'}$$

$$arr[2] \oplus 3 + 0\text{x}30 = 1 \oplus 3 + 0\text{x}30 = 0\text{x}32 = \texttt{'2'}$$

$$arr[3] \oplus 3 + 0\text{x}30 = 2 \oplus 3 + 0\text{x}30 = 0\text{x}31 = \texttt{'1'}$$

$$arr[4] \oplus 3 + 0\text{x}30 = 1 \oplus 3 + 0\text{x}30 = 0\text{x}32 = \texttt{'2'}$$

$$arr[5] \oplus 3 + 0\text{x}30 = 2 \oplus 3 + 0\text{x}30 = 0\text{x}31 = \texttt{'1'}$$

$$arr[6] \oplus 3 + 0\text{x}30 = 1 \oplus 3 + 0\text{x}30 = 0\text{x}32 = \texttt{'2'}$$

$$arr[7] \oplus 3 + 0\text{x}30 = 2 \oplus 3 + 0\text{x}30 = 0\text{x}31 = \texttt{'1'}$$

The password is `21212121`. We can now verify our findings by calling the program with this password as argument and we are actually presented with the correct message telling us we found the flag.

The program also includes one function which is never called, named *secretFunction*, shown in figure 5. This function only prints a single line of text saying `The flag is <<shhimhiding>>`. We can assume that this is a simple misdirection in case we would end our analysis with only extracting strings from the binary.

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

Fig. 5. Decompiled *secretFunction*

V. Side channel attack

After statically analyzing the binary, it is apparent that the provided executable is vulnerable to a side channel attack. The side channel in this case is the number of executed assembly instructions.

Figure 6 shows part of the reverse engineered binary which is responsible for checking if the provided password is correct. In case the program encounters an incorrect character, it sets a flag denoting the password is wrong, and does not do anything otherwise.

```
bool correct = true;
for (int i = 0; i < 8; i++) {
  if ((input[i] - 0x30 ^ 3) != arr[i]) {
    correct = false;
  }
}
```

Fig. 6. Password checking

This approach leaks information about the password checking process as each incorrect character in the input will increase the number of executed instructions by one. Therefore it is possible to find the password one character at a time. This attack is performed by the Python script in appendix A. It retrieves the correct password (`21212121`, as mentioned in section IV) character by character just by starting the program with different inputs and counting the program's instructions.

This type of attack can be mitigated by performing constant number of operations and instructions regardless of the input. The fixed version can be seen in figure 7. It utilizes XOR operation on the two compared values and saves the result in the flag variable. In case all the values of input and control array are the same, the flag is zero. The number of OR and XOR operations is always the same and the flag is now inverted, therefore it signals if the password is wrong, as the name implies.

Side channel attacks are also relevant to the topic of hardware security. Secure hardware such as credit cards and other cryptographic chips need to prevent all kinds of data leakage through side channels, e.g. timing channel, power consumption. If the hardware is not designed in a way that prevents these attacks, it may leak important information, such as encryption keys, thus compromising security of the system.

```
int wrong = 0;
while (i < 8) {
  wrong |= (input[i] - 0x30 ^ 3) ^ arr[i];
  i++;
}
```

Fig. 7. Fixed password checking

VI. Conclusion

In this paper we discussed common reverse engineering techniques along with various software tools which make the task easier, increase productivity and save time for the reverse engineer.

We have fulfilled the goal of the challenge to reverse engineer the provided executable and retrieve the password (flag). We utilized two different approaches.

Firstly, we retrieved the password by statically analyzing the executable, understanding it and reversing its algorithm for checking password.

Secondly, we demonstrated that it is possible to crack the program's secret password using a side channel attack by providing the program with different inputs and subsequently counting the number of executed instructions.

```
import subprocess
import string
import re

def count_instructions(args: list):
    callgrind = ['valgrind', '--tool=callgrind', '--callgrind-out-file=/dev/null']

    res = subprocess.run(callgrind + args, capture_output=True)
    inst_count = re.search('Collected : (\d+)', str(res.stderr, 'ascii')).group(1)

    return inst_count, res

if __name__ == '__main__':
    binary = './qualification.out'
    i = 0
    password = [''] * 8
    prev_inst_count = 0
    first = True
    done = False

    while not done:
        for c in '\x01' + string.printable:
            password[i] = c
            pw = ''.join(password)

            inst_count, res = count_instructions([binary, pw])
            out = str(res.stdout, 'ascii')

            if len(out) > 0 and 'flag' in out:
                print(out, pw)
                done = True
                break

            if first:
                first = False
            elif inst_count < prev_inst_count:
                i += 1
                prev_inst_count = inst_count
                break

            prev_inst_count = inst_count
```

Fig. 8. Python cracker