Reverse a Binary Code with Ghidra

Romain Brenaget, Jérôme Blanchard, Mathieu Dolmen and Pierre Fontaine
Master Cybersecurity of Embedded Systems
Faculté des Sciences et Sciences de l'Ingénieur
Université de Bretagne Sud, Lorient, France
{brenaget.e1803332,blanchard.e1804130,dolmen.e1700417,fontaine.e1800982}@etud.univ-ubs.fr

***Abstract*—This paper aims to demonstrate how a binary code named `qualification.out` can be reversed using a Software Reverse Engineering (SRE) suite of tools developed by NSA's Research Directorate, named Ghidra. First, some preliminary tests have been done on the binary to evaluate it. Then, a static analysis has been realized with Ghidra followed by a dynamic analysis with Angr, both of which have successfully led to find the flag (or secret). Finally, techniques which can be used to dump a firmware and debug it are presented.**

***Index Terms*—Ghidra, reverse engineering, angr, firmware, embedded systems, security**

I. Introduction

The number of embedded devices is significantly increasing, for most of us they are part of our daily lives. By definition, those objects are ubiquitous and are often physically accessible. Thus, we can manipulate and attack them easily through different approaches like software or physical-based attacks. Plus, in most cases, cheap hardware tools are enough to completely exploit a device. Thanks to reverse engineering tools (software and/or hardware) it is quite easy to extract data from an embedded system (e.g. the firmware). Our target is the binary file provided to us by the CSAW ESC organization team. In this paper we will see how we can reverse such a file using Ghidra, an open source tool.

This paper aims to demonstrate four approaches to reverse a binary code and also how a firmware could have been extracted from an embedded device such as an RFID reader.

II. Preliminary tests

When the target program `qualification.out` is executed there is no specific output about its usage. The same thing happens when passing different arguments to the program. By executing the command `strings qualification.out | grep flag` some information can be obtained as shown below:

```
$ strings qualification.out | grep "flag"
Great Job! The flag is what you entered
The flag is <<shhimhiding>>
```

Listing 1. Results from strings command

An output message `Great Job! The flag is what you entered` is telling the success of an input from a supposed argument of the program. It is followed by the given flag `<<shhimhiding>>`. This solution looks too easy but we have to try: `qualification.out <<shhimhiding>>`.

As imagined, the binary did not give the victorious message. We have to dive deeper in the inner working of that binary with static and dynamic analysis.

III. Binary static analysis with Ghidra

A. Gathering information

Ghidra [1] is able to provide a lot of information about the executable (cf. Listing. 2) as it could be done using the `file` command.

```
Program Name: qualification.out
Language ID: x86:LE:64:default (2.8)
Compiler ID: gcc
Processor: x86
Endian: Little
[...]
# of Functions: 22
[...]
ELF File Type: executable
```

Listing 2. Information given by Ghidra

As shown in this output, the binary is a `64-bit x86` executable, using a little endianness. It means that we are going to deal with `x86` assembly instructions with a specific structure. At a first glance there is three important functions which are respectively the `main` function, followed by `challengeFunction` and `secretFunction`. Once the analysis process of Ghidra is launched, a decompiled code is generated from the assembly code. We can now simplify it with the code edition feature.

B. Re-organizing the code

First, we modified the *main* function to make it easier to read.

```
undefined8 main(int iParm1,long lParm2) {
  if (iParm1 == 2) {
    challengeFunction(*(char **)(lParm2 + 8));
  }
  return 0;
}
```

Listing 3. Raw decompiled code

It seems the original code was written in `C++`, so we can modify the signature of the main function. The only thing to do is to change the return type into an `int` and set the two parameters as it would be in a classic `C++` main function (cf. Listing 4). Once those actions are committed, the body of the function is affected and it is easier to read and understand what is happening.

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

Listing 4. Simplified decompiled code

The main function checks if exactly one argument is passed (on Unix systems `argv[0]` is the name of the program) and call `challengeFunction(char* arg)` with the argument passed.

C. Reversing the algorithm

Now, the called function can be analysed, it is noticeable by looking at its decompilated form that an array is hardcoded:

```
Z[8] = [1,2,1,2,1,2,1,2];
```

The goal of this function is to perform a transformation (cf. Listing 5) on each element of the array passed in parameter and compare the result to the local array.

```
while (i < 8) {
    if (((int)param_1[(long)i] - 0x30U ^ 3) !=
        local_28[(long)i]) {
      bVar1 = false;
    }
    i = i + 1;
}
```

Listing 5. Calcul performed on array

Array elements corresponding to the outputs need to be found, those outputs are respectively 1 and 2. Let's call X the vector corresponding to the argument passed and Z the vector matching the key.

$$X_i = (Z_i \oplus \texttt{0x3}) + \texttt{0x30}$$

Fig. 1. Reverse formula

So when $Z_i = 1$ we have $X_i = (\texttt{0x1} \oplus \texttt{0x3}) + \texttt{0x30} = \texttt{0x32}$ and when $Z_i = 2$ we have $X_i = (\texttt{0x2} \oplus \texttt{0x3}) + \texttt{0x30} = \texttt{0x31}$. The ASCII codes matching those two values are respectively 2 and 1, so the flag should be:

```
21212121
```

The final step remains to try it on the binary:

```
$ ./qualification.out 21212121
Great Job! The flag is what you entered
```

Listing 6. Output from the binary using correct flag

D. Scripting with the Ghidra API

Ghidra offers a good API to interact with the binary code [2]. In the case we would have to deal with a more complex program, scripting is a very effective way to ease the process of static analysis and to speed up tedious tasks. We can imagine a scenario were an update of the program is shipped and only the secret has changed (not the logic of the function). In that case we would instantly get the new secret without any additional effort.

The following script (a Java program) demonstrate the use of this API and lead to the same result as described above. Here is the main function named `run` (cf Listing7) of our Java program. It calls different methods, the goal is to slice the script with methods.

```
public void run() throws Exception {
  [...]
  mainFunc = fetchMain();
  [...]
  funcCalledByMain = mainFunc.
    getCalledFunctions(monitor);
  challFunc = (Function) funcCalledByMain.
    toArray()[0];

  code = fetchCode();
  ref_code = code.getSymbol().getReferences();
  code_value = fetchData();
  [...]
  reverseScript();
}
```

Listing 7. run method

First, the `main` function of the binary code is stored in a variable so the content and interactions can be analyzed. The `currentProgram` is a class representing the binary and it provides a method listing all functions in it including the `main()`.

```
private Function fetchMain() throws Exception {
  Listing l = currentProgram.getListing();
  List<Function> main =  l.getGlobalFunctions(
      "main"
  );
  if (main.isEmpty()) {
    throw new Exception(
      "Exception, main function not found"
    );
  }
  return main.get(0);
}
```

Listing 8. FetchMain method

In the `run` method, once the `main` is retrieved using call references, picking the first function called can be done using one API method. This function is affected to the variable `challFunc`. Assuming that `challFunc` is known to perform some treatment on local data, the next step is to find those local data. Note that the decompiled code after treatment had this form (cf. Listing 10), thus the variable to search is named `code` which is an array of 8 elements.

```
  uint code [8];
  [...]
  code[0] = 1;
  [...]
  code[7] = 2;
```

Listing 9. Decompiled challengeFunction

This step consists in iterating over each local variable until finding the one named `code`.

```
private Variable fetchCode() {
  Variable[] localVar = challFunc.
    getLocalVariables();
  Variable code_f = null;
  for (int i = 0; i < localVar.length; i++) {
    Variable var = localVar[i];
```

```
    if (var.getName().equals("code")) {
      println("variable code[] found");
      code_f = var;
    }
  }
  return code_f;
}
```

Listing 10. fetchCode method

Since the values are not stored directly into the memory, the challenge is to find the different instructions that assign a value to each cells of the `code` variable. Using the `getCodeUnitAfter()` method, a result such as in Listing 11 can be produced.

```
csaw.java> MOV dword ptr [RBP + -0x1c],0x2
```

Listing 11. Example of assembly affectation

In the method below, the assembly instruction that moves a value to the array is fetched, and for each of those instruction, values can be isolated by splitting them using the "," as a delimiter to keep only the value affected. Those values are returned in a `Byte` array.

```
private Byte[] fetchData() {
  Byte[] res = new Byte[8];
  for (int j = 0; j < ref_code.length; j++) {
    String res_str = currentProgram.
      getListing().
      getCodeUnitAt(
        ref_code[j].getFromAddress()
      ).toString().split(",")[1];
    Byte res_byt = Byte.decode(res_str);
    res[j] = res_byt;
  }
  return res;
}
```

Listing 12. fetchData method

The last step consists in using the data returned by the previous method and applying a treatment for each value. This treatment (cf Listing 13) is just an implementation of what have been done before (cf : Fig. 1).

```
private void reverseScript() {
  char[] res = new char[8];
  for (int i = 0; i < 8; i++) {
    res[i] = (char) (
      (code_value[i] ^ Byte.decode("0x3")) +
      Byte.decode("0x30")
    );
    print("" + res[i]);
  }
  print("\n");
}
```

Listing 13. ReverseScript method

As a result, our Ghidra script produces the following output:

```
csaw.java> Running...
[...] //Verbose information here
csaw.java> [!] res format ascii
[+] 21212121
csaw.java> Finished!
```

Listing 14. Output of the Java program

Another way to retrieve the flag is this time to perform a behavioral analysis while the binary code is running.

IV. Dynamic analysis

A different approach to solve this challenge (finding the password of the binary) is to use a SAT solver (i.e. Satisfiability Solver) to evaluate which sequence of branches need to be taken in order to get to the "success" message. In a nutshell, a SAT solver converts conditions into boolean equations [3]. A popular framework leveraging this technique is angr [4], which comes as a Python library. The following script is an example of how to use it. We can note its very short length. The power of this technique resides in the fact that it does not require much knowledge of what the program being analyzed is doing, one just needs to identify the offset of an instruction we want to see executed and let the magic happen! More explanation can be found in [5].

```
#!/usr/bin/env python3
import angr
import logging

def solve(base):
    # offset of the function to be analyzed
    offset_fct = 0x52d
    # offset of the message printed when the pin
    # is correct
    offset_success = 0x5be
    p = angr.Project('qualification.out')
    state = p.factory.blank_state(
        addr=base+offset_fct)
    # start the solver
    sm = p.factory.simulation_manager(state)
    sm.explore(find=base+offset_success)
    if len(sm.found) > 0:
        found = sm.found[0]
    else:
        exit()
    # get the result by looking into the memory
    # in the solver state
    memory = found.memory.load(found.regs.rdi,
    128)
    answer = found.solver.eval(memory, cast_to=
    bytes)
    out = answer[:answer.index(b'\x00')]
    print("Hex output: 0x{}".format(out.hex()))
    print("Raw string output: {}".format(out))
if __name__ == '__main__':
    base = 0x400000
    solve(base)
```

Listing 15. Angr script

Once executed, the angr script allows to find the flag:

```
$ ./solution.py
[...]
Hex output: 0x3231323132313231
Raw string output: b'21212121'
```

Listing 16. Angr script output

However this approach requires to be able to instrument the binary. It is not something we can always do with a firmware from an embedded system. In order to do dynamic binary analysis in that case we need to be able to emulate the firmware using QEMU for example. It may not be always possible depending on the target architecture. One of the major difficulty regarding firmware analysis is to analyze it while it is running. We can not place a breakpoint at the entry point of the firmware with traditional debuggers nor even using a

kernel debugger, an hardware level debugger is needed here. So despite being very powerful, dynamic binary analysis can be quite limited when it comes to apply it on very low-level software such as a firmware.

V. Hardware debugging

As we have seen in the previous section, analyzing a firmware while it is running can be quite a daunting task. For that we can use a hardware-level debugger. However such tools (e.g. JTAG-based) require debug ports enabled on the hardware which are disabled before shipping (or should be).

Recent effort has been done toward providing security researcher with tools to assess the security and robustness of a firmware. The Intel's platform security team plan to release the first stable version of their open source tool during the second semester of 2019 (Host-Based Firmware Analyzer). It will integrate open source tools to analyze isolated part of a firmware from an OS. Its main focus is UEFI firmwares, though it may be possible to use it with an embedded system firmware as well.

Another set of tools at our disposal to get an idea of what is happening on the target device is a logic analyzer or an oscilloscope. With such tools we can infer what parts of the architecture is being used (memory, etc.) depending of the inputs passed to the program used by the user. Furthermore, a thorough analysis requires downloading the code executed by the embedded system.

VI. Reading embedded device memories

In this section we will describe how we would extract the firmware of a device. Indeed, in order to reverse in depth an embedded system to discover its secrets or to modify a firmware, it may be necessary to dump the firmware. A dump (or a copy) can be done with a command line utility (e.g. Avrdude [6]), specific programmers, a hardware reverse tool (e.g. Bus Pirate, Hardsploit) or a dedicated debug interface such as JTAG or UART.

If the chip has an accessible debug port, it may allow to read the firmware through that interface. However, most modern chips have security features (like fuses) preventing firmware from being read through the debug interface. These reverse methods can sometimes require to de-solder the chip from the board or introducing glitches into the hardware logic (e.g. manipulating power or clock sources) to find a way to bypass security measures and access different memories like flash, RAM or EEPROM. For example, dumping an Arduino Uno (based on ATMEGA328P) firmware can be done with avrdude:

```
$ sudo avrdude -v -c arduino -p ATMEGA328P \
-P /dev/ttyACM0 -U flash:r:/tmp/dump.hex:i
```

Listing 17. Avrdude command for binary extraction

This tool shows different memory types of a selected target, fuses states (equivalent to read/write protection) and can download all contents of these memories. It is mandatory to have a look to the datasheet of the component, easily found on the Internet, in order to understand what to analyse and reverse.

```
Memory Type Size  Indx Paged  Size   Size Pages
----------- ----- ---- ------ ------ ---- ------
EEPROM          4    0 no       1024    4      0
flash         128    0 yes     32768  128    256
lfuse           0    0 no          1    0      0
hfuse           0    0 no          1    0      0
efuse           0    0 no          1    0      0
lock            0    0 no          1    0      0
calibration     0    0 no          1    0      0
signature       0    0 no          3    0      0
```

Listing 18. Part of avrdude output

At this step and for a proper exploitation with some disassembly tools, the dump file in .hex format can be converted in a .bin format:

```
$ avr-objcopy -I ihex /tmp/dump.hex -O binary \
/tmp/dump.bin
```

Listing 19. .Hex to .Bin conversion

Once data have been dumped and after proper conversion, they can be analyzed and reversed with tools like Ghidra, IDA, Radare2 or some specific softwares depending on the architecture of the embedded system. The machine code can be disassembled by reverse engineer the extracted firmware.

Here, Arduino Uno is based on an ATmega328 microcontroller from the 8-bit AVR microcontroller family, so with an adapted disassembly tool like vAVRdisasm [7] which is an 8-bit Atmel AVR firmware disassembler, the firmware dumped before can now be reversed. The instruction set summary part of the datasheet is very useful at this point.

From there, several implementations can be reversed, for example around Arduino RFID applications. Researched tag (secret) can be compared with a hardcoded string, or directly compared byte to byte with an array (referenced tag, or another implementations). Analyze assembly code or just try one command on a given dumped firmware can sometimes revealed secrets easily (e.g. `strings` command for hardcoded string like RFID tag value). Please notice that Linux commands like `strings`, `readelf`, `file`, `strace` are very efficient to gain information before sinking into the assembler.

VII. Conclusion

We demonstrated how both static and dynamic binary analysis can help to solve this kind of challenge. We also explored techniques that we would use in real condition on a device and mentioned the limitations we could face. All of these approaches showed that the firmware of an embedded system is a very critical part and must be carefully protected. As long as an attacker is able to get his hands on it, the secrets of the device are at risk.

References

[1] "Ghidra." https://github.com/NationalSecurityAgency/ghidra
[2] NSA, "Ghidraclass." https://ghidra.re/courses/GhidraClass/ AdvancedDevelopment/GhidraAdvancedDevelopment.html
[3] "Sat solver." https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-hacklu-2018-samdb-z3-final.pdf
[4] "angr." https://docs.angr.io/
[5] "angr example." github.com/mwrlabs/z3_and_angr_binary_analysis workshop/blob/master/angr/hello_world/generate_serial_solution.py
[6] "avrdude." https://www.nongnu.org/avrdude/
[7] "vavrdisasm." https://github.com/vsergeev/vavrdisasm