Computer Systems
baiCOSY06, Fall 2017
Lab Assignment : Defusing a Binary Bomb
Assigned: September 14, Due: Sept. 23, 18:00.

Lennart Beekhuis, Bram Otten

September 23, 2017

1 Introduction

The goal of this lab is threefold. First, we aim to demonstrate how a debugger can and should be used to determine and/or correct the *actual* functionality of a piece of code (e.g., a procedure or a loop). Second, we aim to learn more about low-level programming. Therefore, we use gdb as a debugger and assembly code as the program under investigation. Third, and final, we aim to demonstrate the use of debugging for reverse engineering applications where there is no source code to analyze. In turn, such a reverse engineering approach reveals enough about the code to enable automated testing and performance analysis - although these are not our concerns for now.

The lab is setup as follows: consecutive ”bombs”, of increased difficulty, are combined into an executable program. In each phase, a specific input ”code” is required in order for the program not to crash. A crash is, in turn, equivalent with a bomb ”explosion”. The goal is to determine all 6 codes correctly, and thus finish the program ”peacefully”. With the help of a disassembler, we can obtain the assembly code of each phase. Using the debugger on this low-level code, we reverse engineer each phase and determine correct codes to defuse each phase.

This report aims to describe the process of reverse engineering based on gdb and assembly reading. Specifically, we will describe the general approach, the challenges in each phase, and reflect on the lessons learned.

2 Background

In this section we briefly introduce the necessary background and tools used for completing his lab.

Debugger

A debugger is a tool (i.e., a program in this case) that help identifying and solving correctness problems in applications.

Debuggers help by allowing a controllable execution of the program, potentially with different kinds of inputs, and offering tools to inspect intermediate results. In turn, this flexibility allows the expected and actual behaviors of the target code to be compared and, in case of a mismatch, the functionality can be corrected.

In this lab, we have used the gdb debugger. Specifically, the functions we have used the most are:

Q1: Please enumerate the most important 5 ”functions” of the debugger (i.e., gdb in this case) **you** have used to solve this lab. If you have used less than 5 functions, enumerate all of them. For each function, please briefly explain what it is doing. You can use a table or an enumeration. For example: disassemble is such a function, info is another.

1. ”break”
   stops the program at a certain instruction.
2. ”run”
   runs the program until it encounters a breakpoint (or until the program is done running)
3. ”disassemble”
   shows the current block of code the program is running and which instruction the program counter is currently at.
4. ”set”
   set the value of a memory address or a register.
5. ”x/d”
   interprets the data of a register/memory address as an integer and shows it.

Assembly code

We are using the assembly code to trace the execution of the application without having access to source code. Effectively, this means that we observe the code generated by the compiler and infer the properties of the original application. Moreover, we attempt to reconstruct as much as possible from the original code of interest.

The main challenges in using an approach based on assembly code are: Q2: What are the challenges of using reverse engineering based on assembly (i.e., is this harder than source code and why)? List and briefly explain 3 examples of assembly constructs you found among the most difficult ones. What was the difficulty?

For loops were probably the hardest to recognize. The difficulty wasn't recognizing the loop however. The biggest problem was recognizing what exactly was in the looping code block. Especially when we had to deal with for-loop-ception, it gets really cluttered really fast and recognizing which registers are essential for which loop is hard.

3 General approach

In this section we describe the general strategy we have devised to approach solving the bomb phases, and the main challenges we have identified early on.

Q3: Please describe the general approach you took/wanted to take for defusing the bombs (i..e, how did you go about reverse engineering with gdb?). Were there any immediate challenges emerging from this strategy?

At first, we tried to break down the assembly code into easier, logical statements. Then, we tried to find connections between statement, like a known value (f.e. out input) getting passed on from one register to another. Our first priority was almost always trying to find where our input got stored, so we could see what it got compared to/changed into. If we could find our input, we'd study what the code did with our input and most of the time, we'd recognize some kind of loop or pattern. This approach generally worked well.
If we couldn't find our input, we'd use a combination of breaking down code (rewriting it into more simple pseudocode) and trial and error to hopefully get one step further. This approach took a lot more time though, because not knowing on what the code operates makes it hard to judge what your answer is really good for.

4 Phase-by-phase analysis

We present here a detailed analysis of each of the defuse phases. For each phase, we present its specific challenges and the "reconstructed pseudo-code" - i.e., one of the possible forms of the original code that has generate that assembly structure.

For example, check the following assembly code:

```
   mov    $0x0,-0x4(%rbp)
   mov    $0x1,-0x8(%rbp)
   jmp    .L2
.L1:
   add    $0x5,-0x4(%rbp)
   add    $0x1,-0x8(%rbp)
   cmp    $0x4,-0x8(%rbp)
   jle    .L1
.L2:
   mov    -0x4(%rbp),%eax
```

One of the possible forms of the original code is:

```
a = 0;
for (b=0; b<=4; b++) a = a + 5;
return a;
```

Phase 1

Q4: Please present the "reconstructed pseudo-code" computation of this phase *after* reverse engineering. Please explain what were the specific steps taken to identify it, focusing on the actual values of the parameters (which eventually build the solution). Finally, please present the solution itself.

```
string = input;

if (string =/= "I'm the mayor. I can do anything I want.")
explode_bomb;

return;
```

Phase 2

Q5: same as Q4.

```
6_numbers = input;

if (6_numbers =/= actually 6 numbers)
explode_bomb;

if (6_numbers[0] =/= 1)
explode_bomb;

for (i = 1; i < 6; i++)
    if 6_numbers[i] =/= 6_numbers[i-1]
    explode_bomb;

return;
```

Phase 3

Q6: same as Q4.

```
number_letter_number = input;

if (number_letter_number =/= actually a number, then a letter, then a
    number)
explode_bomb;

switch(number_letter_number[0]) {

/* basically every case statement looked somewhat like this, so we're
    condensing it into 1 statement */

case (1 2 3 4 5 6):
     if (number_letter_number[1] =/= certain_letter)
     explode_bomb;
     (break;)

default:
     explode_bomb;
}

if (number_letter_number[2] =/= certain_number)
     explode_bomb;

return;
```

Phase 4

Q7: same as Q4.

```
if (input.format != "integer integer") {
    explode();
}

int first = input[firstInteger];
if (first < 0 or first > 14) {
    explode();
}

if (func4(first) != 31) {
    explode();
}

if (input[secondInteger] != 31) {
    explode();
}

return 0; // no error ; defused. (If input was 13 31.)

int func4(int x) {
    y = *x; // the adress of something anyways
    x >>= 1;
    if (x <= y) {
        if (x < y) {
            y = func4(x - 1);
        }

        if (x >= y) {
            return x + y;
        }
    }
}
```

Phase 5

Q8: same as Q4.

```
void phase_5(string input) {

    if (input.length != 6) {
        explode();
    }

    int map[5] = [2, 10, 1, 12, 16]; // the 12 was found with x/d $rdx+4*4
    int total = 0;
    for (int i = 0; i < 6; i++) {
        total += map[input[i]];
    }

    if (total != 61) {
        explode();
    }

    // return without error if input was 555103.
}
```

Phase 6

Q9: same as Q4.

```
void phase_6(string input) {

    if (input.format != "integer integer integer
                         integer integer integer") {
        explode();
    }

    for (int i = 0; i < 6; i++) {
        if (input[i] > 6) {
            explode();
        }

        if (input[i] in input twice) {
            explode();
        }
    }

    // It was a bunch of nodes instead of an array.
    int map[6] = [354, 136, 105, 880, 677, 733];
    int a, b;
    for (i = 0; i < 6; i++) {
        a = input[i-j];
        b = map[i];
        if (a < b) {
            explode();
        }
    }

    // return without error if input was 3 1 2 6 5 4.
}
```

5 Summary and Future Work

In this lab we have learned to use a debugger, practiced our assembly reading skills, and build a strategy to use a debugger for code reverse engineering.

Lessons Learned

Q10: What have you learned during this lab? Focus on your own knowledge and experience, and new skills (i..e, avoid "standard" statements). Consider both lab components: the debugger and the assembly.

I think learning how to translate a language like assembly into (pseudo)code is a skill that's very valuable to have. It's such a different skill compared to writing a program, and it's the first time i've had to do this in these 1.5 years studying AI. So that's probably the biggest thing I've learned. Besides that, i feel inclined to put a lot of standard statements here: learning assembly, learning how to work with gdb, etc.

The most important "standard statement" is probably learning to work with a debugger, as that's also something i feel will come in handy later in our careers. You're gonna have to debug code virtually everywhere, and f. e. if you want to really optimize code, you'll have to debug in the same way we did for this assignment (very low level, so you can see exactly what's going on).

Future work

One of the missing parts in this lab has been the validation of our reconstructed code.

Q11: How would you validate that your reconstructed pseudo-code for any phase is correct? Can the process be automated?

It couldn't be automated, as everyone has a different way of writing pseudocode. However, if the pseudocode was actual, correct C code, you could automate the verification process. Either by comparing the C code or by comparing the assembly code of the 2 programs.

Q12 (optional): Any other ideas you would like to add here?

We spent about 4 hours on the first phase. In hindsight, i feel like we were pretty dumb to get stuck for 4 fucking hours at such a simple piece of code, but i've heard of other people who also got stuck at phase 1, so maybe it's a good idea to release some bigger hints or just explain the whole phase in the 2nd or 3rd werkcollege, because it feels like wasted time. Maybe you could make phase 1 into phase 0, and add a newer, somewhat harder phase 1 to give a more gradual introduction.