Computer Systems
Fall 2018
Lab Assignment : Defusing a Binary Bomb
Assigned: September 12, Due: Sept. 21, **20:00**.

Tim Stolp, Sybren Osinga

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. A debugger works by firstly catching the execute syscall and blocking the start of the execution. Then it queries the CPU registers to get the process' current instruction and stack location. After

that it catches for clone/fork events to detect new threads. And finally it reads data addresses to read and alter memory variables.

Q0 (optional): How is a debugger implemented?

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 (1.0p): 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 (function)
   This causes the program to stop before executing that function while running the code. (to be able to look into the assembly code of the function).

2. layout asm
   This shows a layout in which you can see at which step in the assembly you are and by using the ”nexti” command you can step through the assembly code 1 by 1 while updating where you are real time.

3. disas
   This shows the assembly code of the function you are at at that moment.

4. print
   This prints out the information at a certain memory address or register.

5. i r
   This prints all the values of all registers at that moment.

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:

1. Figuring out if a jump condition is going to be true or false because you need extensive knowledge of the compare functions and how they set the flags and then also what flag conditions are necessary to make the jump condition true.

2. It is almost impossible to track what value is in what register or memory address without constantly printing the value because the names of the registers or address numbers give no information about the value behind them.

3. Assembly code has a lot of lines of codes that have barely any influence on the main workings of the program. (for example mov because its not too important to know what exact register the info goes into, but it's more important what the program does with that information in the register.)

Q2 (1.0p): 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?

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.
Our general approach was first getting the objdump of the file, then setting a breakpoint at the function we want to look at, run the code with a test input, look at the assembly code of the function using 'disas' and try to figure out what it does in general and especially look at what conditions have to be met to jump over the bomb call function, afterwards we step through the code of the functon with 'layout asm' and 'nexti' printing out various registers and memory addresses to find which one contains our input and then look at what is done with that input.

Q3 (1.0p): 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?

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

He is evil and fits easily into most overhead storage bins.
It compares your input to a certain string.
print out various things until I found 0x402260 which contained the string.

Q4 (1.0p): 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.

Phase 2

1 2 4 8 16 32
it checks if first number is equal to 1, afterwards it adds itself to itself and compares again to the next number and loops this until 6 numbers are checked.

Q5 (1.0p): same as Q4.

Phase 3

2 518
First figure out what I need to input by x/s 0x40246f,
then I found that the first int needs to be bigger than 1 and lower than 7,
I took 2, then it did a couple of artheritmic adds and subs which added up to
0x206 which was then compared to my 2nd int if they are equal,
so the 2nd int was 518 in decimal

Q6 (1.0p): same as Q4.

Phase 4

4 19 DrEvil
first it checks if there are 2 numbers as input then if the first number is lower than 15,
then in func 4 it compares the first number first to 7 then if 7 is higher it pushes 7 to the stack and compares the first number to 3
then because 3 is lower it pushes 3 onto the stack and compares the first number to 5,

then because 5 is higher it pushes 5 onto the stack and compares the first number to 4,
then it is equal so it returns the value which is the first number + the numbers it pushed onto the stack,
here 4+5+3+7=19 then it checks if returned number is equal to 19 and if second input number is same as 19.
(add DrEvil to access secret phase, stepping through sscanf showed that it was looking for ”%d %d %s” during the call in the last phase_defused.)

Q7 (1.0p): same as Q4.

Phase 5

bbbbbi
string length gets counted needs to be 6, goes in recursion and adds a certain number per iteration depending on which letters you input,
input a bunch of letters and write down their respective number, total number needs to be 37, b=6,i=7, 5x6 + 1xi = 37.

Q8 (1.0p): same as Q4.

Phase 6

6 4 3 5 1 2
first it checks if 6 numbers are put in, then it checks if all ints are below 6,
then if all ints are all different from eachother,
then it compares numbers from $rbx to eachother.
the numbers compared can be found by printing $rbx by x/w which shows that each number belongs to a certain node, nodes 1 to 6,
the values are the values connected to that node, and so to that number of the node.
Sort the values from the nodes from big to small and put numbers 1 to 6 in order of their respective value.

Q9 (1.0p): same as Q4.

Phase 7 (secret phase)

35
to reach fun7 your 7th input in the solution.txt file must be: input-1 =¡ 1000,
the result of fun7 is compared to 6, if equal phase gets defused. fun7 has fixed numbers that it compares to the input, if the number is greater than the input it does the return value *2+1, else it does *2, if equal it returns 0.
this creates a tree structure and depending on which way you go in the tree, you do either *2+1 or *2,
to get 6 you need to do 0*2+1=1, 1*2+1=3, 3*2, so starting backwards, less than -¿ greater than -¿ greater than -¿ equal gets you the right calculations.
following these steps you end up with 35.

Q12 (?p): same as Q4.

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

During this lab we've become more accustomed with how the value of a memory address and the address itself are 2 different values and are better at distinguishing them in assembly code. Next to that we had barely any knowledge on debuggers before so most of what we've done and learned are new skills, for example how to use break points and how to go through the code step by step. Also it's become more clear how exactly if statements are handled on a deeper level.

Q10 (1.0p): 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.

Future work

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

One way to validate our pseudo-code is to turn it into actual code and give the original and reconstructed program the same various inputs and see if all the outputs are identical. This could probably be automated by having another program call the 2 other programs together with test solution.txt files with different inputs and per call of the 2 programs per phase return 1 if identical and 0 if not, then either print those results or put them in a list and print that. For example each element in the list is a different test solution.txt file, and each element of that list is a list of 6 values of either 1 or 0 representing the 6 phases.

Q11 (1.0p): How would you validate that your reconstructed pseudo-code for any phase is correct? Can the process be automated?