Practical Session – Week 8

Objectives

1. Interpret and manipulate assembly code via hardware debugging techniques
2. Apply reverse engineering techniques to identify main software flaws

Basic commands on linux

1. *objdump -d binary.file : show all assembly*
2. *strings binary.file : show all strings*
3. *gcc file.c -o binary.file -g -O0/3*
4. *gcc file.c -S assembly.s -O0/3*

Basic commands on gdb

Please see this link https://sourceware.org/gdb/current/onlinedocs/gdb/

1. set disassembly-flavor intel : show intel syntax instead of AT&T
2. break or b : set a break point
   - b main : break to main function
   - b *0x0342FA0230 : break to this program address
3. run : goes to the first breakpoint
4. continue : run/go to the next breakpoint
5. return : step out of the function by cancelling its execution
6. si : Execute one machine instruction, then stop and return to the debugger
7. x/s : show the content of specific memory address
   - x/s 0x402400 or x/s $rax
8. info registers or i r : show the content of the registers, e.g., i r $rip shows the next instruction to be executed (%rip register holds the next instruction)
9. disas : show the assembly code at this point, or use 'disas function1' to display the assembly of this function
10. print : display individual register value
    - print /d $rax : display the value of rax register in decimal
    - print /t $rax : display the value of rax register in binary
    - print /x $rax : display the value of rax register in hexadecimal
11. The "x" command is used to display the values of specific memory locations: "x/nyz"
    - "n" is the number of fields to display
    - "y" is the format of the output, 'c' for character, 'd' for decimal and 'x' for hexadecimal
    - "z" is the size of the field to be displayed, 'b' for byte, 'h' for 16-bit word, 'w' for 32-bit word
    - 'x/10xw $rsp' : displays in hex first 10 32-bit contents of the stack

Task – Bomb Lab, try to defuse the Bomb

This week you will reverse engineer the bomb lab game and try to defuse the bomb. This is an original source of exercise from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Carnegie Melon University. You can find more information about this task in http://csapp.cs.cmu.edu/public/labs.html.

This game consists of 6 phases. In each phase, you must enter the right password otherwise the bomb explodes. ***In this practical, you will defuse just the first phase***.

How to run it: You can just type *'./bomb'* or type *'./bomb inputs.txt'*, where in '*inputs.txt*' the input strings are.

How to start defusing the bomb: First, you must use "*objdump –d bomb*" command to generate its assembly. You can use "*objdump –d bomb > output.txt*" command to redirect the output to a .txt file. You can also run the command "*strings bomb > output2.txt*" to see all the strings of the binary; a password might be stored there, which is a serious software flaw (actually it does). Part of the C-code is also provided in bomb.c file, to better understand the structure of the code. After you have had an idea of the program's functions, it is time to start reverse engineering phase1 routine (in this practical you will defuse only the first phase of the game). Can you identify where the input message is read? Use gdb and start studying phase1 routine step by step trying to understand what the code does. Make sure you understand what every instruction does.

First, we need to identify the part of the code where the input is read. In main() before phase_1() routine, there is a read_line() function call, see below:

```
  0x0000000000400e2d <+141>:   callq  0x400b10 <puts@plt>  //a message is printed on the screen
  0x0000000000400e32 <+146>:   callq  0x40149e <read_line>  //our input is taken
=> 0x0000000000400e37 <+151>:   mov    %rax,%rdi             //rax contains the output of read_line().
This value is put into rdi in order to be passed to phase_1()
  0x0000000000400e3a <+154>:   callq  0x400ee0 <phase_1>  //call phase_1
```

Note that the output of each function is always stored into %rax register. Thus, if we put a breakpoint at 0x0000000000400e37, by using the following command 'b *0x0000000000400e37' and then 'continue', then, we can check the status of the %rax register. So the steps are as follows

*b *0x0000000000400e37*
continue
i r rax

The content of the rax is '0x603780'. This is a hex number which is the memory address of where our input is stored. We can check this memory address contents by using

x/s 0x603780

Then rax is stored into %rdi and phase_1() is called (why?). **When a function is called, its operands are always stored into (*rdi,rsi,rdx,rcx,r8,r9*).** In this case, it looks like phase_1() takes just a single input operand (%rdi, i.e., our input).

The next step is to check phase_1() routine by using the following commands

b phase_1
continue
disas

The assembly of phase_1() follows (I have included comments to better undertands what it does):

```
Dump of assembler code for function phase_1:
=> 0x0000000000400ee0 <+0>:    sub    $0x8,%rsp               //allocates 8 bytes to the stack
   0x0000000000400ee4 <+4>:    mov    $0x402400,%esi          //puts something to esi (why? esi is the second operand of the following function. the first is rdi, which is our input)
   0x0000000000400ee9 <+9>:    callq  0x401338 <strings_not_equal>   //calls a function to compare our input to the secret input (not that hard to figure that out)
   0x0000000000400eee <+14>:   test   %eax,%eax   //eax AND eax, jump below if the %eax is zero. If two operands are equal, their bitwise AND is zero when both are zero
   0x0000000000400ef0 <+16>:   je     0x400ef7 <phase_1+23>  //if the output of the function is zero continue to 0x0000000000400ef7
   0x0000000000400ef2 <+18>:   callq  0x40143a <explode_bomb> // else explode the bomb
   0x0000000000400ef7 <+23>:   add    $0x8,%rsp               //deallocates 8 bytes from the stack
   0x0000000000400efb <+27>:   retq
```


So, using the following commands we can see the input operands of strings_not_equal()

*b *0x0000000000400ee4 (or use si)*
continue
i r rdi
x/s 0x603780 (the content of rdi). This will give our input string
x/s 0x402400 (the content of esi). This will give the desired string.

Although, we have solved phase_1(), we will continue reverse engineering the code to understand what it does. The assembly code of *strings_not_equal()* follows. This routine calls the *string_length()* routine twice (why? this is suspicious). In the first time it outputs the length of our input, while in the second it outputs the length of the secret string. If you check the value of %eax register after the function is called, you will figure this out. Please do so. *string_length()* takes only one operand as input, just %rdi; this can be justified by checking *string_length()* assembly.

```
Dump of assembler code for function strings_not_equal:
=> 0x0000000000401338 <+0>:    push   %r12
   0x000000000040133a <+2>:    push   %rbp
   0x000000000040133b <+3>:    push   %rbx
   0x000000000040133c <+4>:    mov    %rdi,%rbx
   0x000000000040133f <+7>:    mov    %rsi,%rbp
```

```
0x0000000000401342 <+10>:   callq  0x40131b <string_length> //takes %rdi as input (our input)
0x0000000000401347 <+15>:   mov    %eax,%r12d //put the length of our input to %r12
0x000000000040134a <+18>:   mov    %rbp,%rdi  //put the secret message to %rdi to pass it to the string_length()
0x000000000040134d <+21>:   callq  0x40131b <string_length> //takes %rdi as input (secret message)
0x0000000000401352 <+26>:   mov    $0x1,%edx   //this is the output value of strings_not_equal()
0x0000000000401357 <+31>:   cmp    %eax,%r12d //compare the length of the secret message with the input's
0x000000000040135a <+34>:   jne    0x40139b <strings_not_equal+99>  //if eax==r12, then ZF=1 and thus jump below and successfully exit the function
0x000000000040135c <+36>:   movzbl (%rbx),%eax
0x000000000040135f <+39>:   test   %al,%al
0x0000000000401361 <+41>:   je     0x401388 <strings_not_equal+80>
0x0000000000401363 <+43>:   cmp    0x0(%rbp),%al
0x0000000000401366 <+46>:   je     0x401372 <strings_not_equal+58>
0x0000000000401368 <+48>:   jmp    0x40138f <strings_not_equal+87>
0x000000000040136a <+50>:   cmp    0x0(%rbp),%al
0x000000000040136d <+53>:   nopl   (%rax)
0x0000000000401370 <+56>:   jne    0x401396 <strings_not_equal+94>
0x0000000000401372 <+58>:   add    $0x1,%rbx
0x0000000000401376 <+62>:   add    $0x1,%rbp
0x000000000040137a <+66>:   movzbl (%rbx),%eax
0x000000000040137d <+69>:   test   %al,%al
0x000000000040137f <+71>:   jne    0x40136a <strings_not_equal+50>
0x0000000000401381 <+73>:   mov    $0x0,%edx
0x0000000000401386 <+78>:   jmp    0x40139b <strings_not_equal+99>
0x0000000000401388 <+80>:   mov    $0x0,%edx
0x000000000040138d <+85>:   jmp    0x40139b <strings_not_equal+99>
0x000000000040138f <+87>:   mov    $0x1,%edx
0x0000000000401394 <+92>:   jmp    0x40139b <strings_not_equal+99>
0x0000000000401396 <+94>:   mov    $0x1,%edx
0x000000000040139b <+99>:   mov    %edx,%eax   //put in eax the output of the function
0x000000000040139d <+101>:  pop    %rbx
0x000000000040139e <+102>:  pop    %rbp
0x000000000040139f <+103>:  pop    %r12
0x00000000004013a1 <+105>:  retq
```