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:

Text

Description automatically generated

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.

Text

Description automatically generated

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

A picture containing text

Description automatically generated

Disas add\_bash - 0x0804920f

Text

Description automatically generated with medium confidence

Disas exec\_string - 0x08049182

Text

Description automatically generated

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

Text

Description automatically generated

Pop ebp addr - 0x0804933b

Pop edi ; pop ebp addr - 0x0804933a

Pop esi ; pop edi ; pop ebp - 0x08049339

Text

Description automatically generated with medium confidence

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. Text

Description automatically generated

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.

Text

Description automatically generated

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.