

COMPSCI 390R

Shellcoding

Important Info:

1. Project 1 due Tuesday midnight
2. Homework 2 is out still
 - Please put effort into the survey, can actually have an effect on the class
3. Project 2 out tonight/tomorrow
 - Will be setting up a remote server for you to connect to and exploit, that might come within the next week

Last Lecture:

1. Explored corrupting the stack memory
2. Learned how to control local variables
3. Managed to redirect code execution to another function in the program

Can we get more though?

The program for the day:

```
#include <stdio.h>

int main() {
    char buf[128];

    printf("%p\n", buf);

    gets(buf);

    return 0;
}
```

This is all you get for today
Seriously!

Is this enough though?

So what do we get?

```
#include <stdio.h>

int main() {

    char buf[128];

    printf("%p\n", buf);

    gets(buf);

    return 0;

}
```

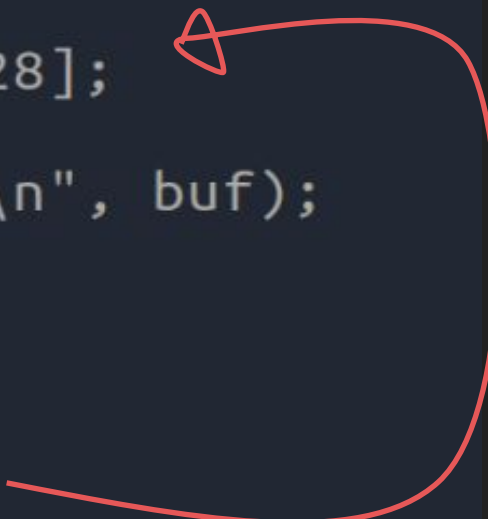
1. A buffer we control
2. Stack/Ret Addr overflow
3. Pointer to the buffer

Any ideas on what we can do?

Shellcode!

```
#include <stdio.h>

int main() {
    char buf[128];
    printf("%p\n", buf);
    gets(buf);
    return 0;
}
```



Since all memory is treated equally, we can write assembly instructions in the buffer!

If we overflow the buffer and make the function return to the start of the buffer, we can then have arbitrary code execution and run a shell

How can we add code to our payload?

Pwntools has a built-in function `asm` which will generate the bytecode of the given assembly instructions. You must set the context first!

```
>>> from pwn import *
>>> context.clear(arch='amd64')
>>> asm("""
... mov rdi, 0
... mov rsi, 0
... """)
b'H\xc7\xc7\x00\x00\x00\x00H\xc7\xc6\x00\x00\x00\x00'
```

What can we make our code do?

- Best thing we can do is pop a shell and gain access to the machine
- Can read/write another file on the machine that we may not have access to otherwise
 - The program we are exploiting may have different permissions than the user executing it
- Can inject and hijack the program to run our own software
 - Examples can be crypto-miners or a C2 listener

What we're going to do in class

```
execve("/bin/sh", 0, 0)
```

Running this line (the assembly equivalent) will pop a shell and give us everything we could ever want.

Turning this into assembly:

- Execve is a syscall in Linux, so we need to learn how to call syscalls in assembly
- We need to setup the parameters of the syscall
 - This includes having the string “/bin/sh” in memory
 - Setting up the registers of the other parameters

Syscalls in Assembly:

- Syscalls are necessary for operations that the userland program can't be trusted to complete.
- Calling a syscall gives up control to the kernel until the operation is complete and the kernel returns control to the userland program

Syscalls are triggered with the opcode “syscall”

Syscalls in Assembly:

- RAX determines the operation to perform when triggering a syscall
- The other typical registers carry parameters for the syscall function

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			

- https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Execve Syscall:

59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			
----	------------	-------------------------	-----------------------------	-----------------------------	--	--	--

- RAX value is 59
- RDI needs to point to /bin/sh
 - We'll need to push this string to the stack, null-terminated
- RSI and RDX we can set to 0 (NULL)

Improving our Shellcode:

- The size of our shellcode matters. We don't always get a large buffer to write into so the less bytes possible the better
- Usually we can't have certain values in our shellcode due our method of input, we must change up the assembly to avoid these pitfalls

Making our shellcode smaller

- We can replace opcodes with alternative but equal operations
 - `mov rdi, 0` with `xor rdi, rdi`

or you can use opcodes such as “inc” (increment) if you know you have a close value already

- Pushing a value and then popping it right away versus `mov` can also save a few bytes

Play around with this to see how small you can get it!
(Our current know limit is 22)

Avoiding certain values in our Shellcode:

Depending on how we get to input our shellcode, we'll need to avoid certain bytes

- Strcpy: Ends on a Null byte \x00
- Scanf, gets, getline, fgets: Ends on a newline \x0a
- Scanf: Ends on a carriage return \x0d
- Scanf: Ends on a space \x20
- Scanf: Ends on a tab \x09

How can we avoid this?

Always look for alternative opcodes

- Moving 0 into a register will contain null bytes, xor with itself will not
- If we want a null byte at the end of a string we can put a dummy value and then shift left and right

Advanced Goals:

Read a file if we can't get a shell

- Use open, read, write syscalls to open a file and write to stdout

Have a machine connect back and send us a shell if we can (useful if firewalls are blocking incoming connections)

- Use the socket, connect syscalls to open a TCP connection
- Use dup2 to redirect file inputs and outputs to the new socket
- Spawn a new shell using execve