

KringleCon 4: Calling Birds!

6) Shellcode Primer

1. Click the **Map** (Destinations) icon and then click **Jack's Office**
2. Click **Shellcode Primer**

Shellcode Primer

Home

1. Introduction **New**

Welcome to Shellcode Primer!

This is a training program conceived by Jack Frost (yes, THE Jack Frost) to train trolls how to build exploit code, from the ground up. This will teach how to write working x64 shellcode to read a file and print it to standard output!

If you're new to this, we recommend reading this introduction thoroughly!

Introduction

In this challenge, you will be hand-crafting increasingly complex shellcode, written in x64. If that sounds scary, don't fret! We will guide you step by step!

Choose your challenge on the left (Introduction will be open by default), read the instructions on the top, and start writing code! We'll provide the basic structure of the code to help make sure you're heading in the right direction.

What is Shellcode?

Shellcode is small, position-independent assembly code that is typically executed as the payload of an exploit. For the initial challenges, you'll write code and see what it does - no exploit required.

The important thing about shellcode is that it doesn't typically have access to libraries or functions that you might be accustomed to; it needs to be entirely self-contained! Even normally simple things like defining a string or opening a file can be tricky. We'll cover those things as they come up!

Using Shellcode Primer

As you type code, it will be assembled in the background. Assembling takes the assembly code you write and translates it into machine code (which is represented as a series of hex characters). We use the `metasm` Ruby library to assemble, in case you want to work on your code locally:

```
require 'metasm'
assembled = Metasm::Shellcode.assemble(Metasm::X86_64.new, payload['code']).encode_string.unpack('H*').pop()
```

When your code successfully assembles, you can execute it by clicking the **Execute** button at the bottom. That'll run the code in a virtual machine, and instrument each step so you can see exactly what's going on!

Good Luck!

3. Complete challenge 1 to 10
4. Click **11. Reading a File**

Shellcode Primer

Home

1. Introduction ✓

2. Loops ✓

3. Getting Started ✓

4. Returning a Value ✓

5. System Calls ✓

6. Calling into the Void ✓

7. Getting RIP ✓

8. Hello, World! ✓

9. Hello, World! ✓

10. Opening a File ✓

11. Reading a File

Reading a File

Do you feel ready to write some useful code? We hope so! You're mostly on your own this time! Don't forget that you can reference your solutions from other levels!

For this exercise, we're going to read a specific file... let's say, `/var/northpolesecrets.txt`... and write it to stdout. No reason for the name, but since this is Jack Frost's troll-trainer, it might be related to a top-secret mission!

Solving this is going to require three syscalls! Four if you decide to use `sys_exit` - you're welcome to return or `exit`, just don't forget to fix the stack if you return!

First up, just like last exercise, call `sys_open`. This time, be sure to open `/var/northpolesecrets.txt`.

Second, find the `sys_read` entry on [the syscall table](#), and set up the call. Some tips:

1. The file descriptor is returned by `sys_open`
2. The buffer for reading the file can be any writable memory - `rsp` is a great option, temporary storage is what the stack is meant for
3. You can experiment to find the right `count`, but if it's a bit too high, that's perfectly fine

Third, find the `sys_write` entry, and use it to write to stdout. Some tips on that:

1. The file descriptor for stdout is always 1
2. The best value for `count` is the return value from `sys_read`, but you can experiment with that as well if it's too long, you might get some garbage after that's okay!

Finally, if you use `rsp` as a buffer, you won't be able to `ret` - you're going to overwrite the return address and `ret` will crash. That's okay! You remember how to `sys_exit`, right? :)

For an extra challenge, you can also subtract from `rsp`, use it, then add to `rsp` to protect the return address. That's how typical applications do it.)

Good luck!

```
; TODO: Get a reference to this
db '/var/northpolesecrets.txt',0

; TODO: Call sys_open

; TODO: Call sys_read on the file handle and read it into rsp

; TODO: Call sys_write to write the contents from rsp to stdout (1)

; TODO: Call sys_exit
```

Request Hint [0 / 1] - Hints are free!

Reset

Execute

Assembles to: 2f766172296ef727466709f6c5736563726574732e74787400

5. Click the **Request Hint [0/1]** button

Request Hint [1 / 1] - Hints are free!

- This is the last challenge. Good luck!

6. Modify the contents of the code window to

```

; TODO: Get a reference to this
call secret_ref
db '/var/northpolesecrets.txt',0
secret_ref:
pop rbx
; TODO: Call sys_open
mov rax, 2
mov rdi, rbx
mov rsi, 0
mov rdx, 0
syscall

; TODO: Call sys_read on the file handle and read it into rsp
mov rdi, rax
mov rax, 0
mov rsi, rsp
mov rdx, 150
syscall

; TODO: Call sys_write to write the contents from rsp to stdout (1)
mov rax,
mov rdi, 1
mov rsi, rsp
mov rdx, 150
syscall

; TODO: Call sys_exit
mov rax, 60
mov rdi, 99
syscall

```

7. Click the **Execute** button

Exit code
Process exited cleanly with exit code 99

Stdout
Secret to KringleCon success: all of our speakers and organizers, providing the gift of cyber security knowledge, free to the community.

Success!
Great work! You just wrote some real life shellcode for reading a file!

Did you know that you can add `?cheat` after the URL (before the `#`) to unlock our solutions?

History

```

0x13370000 call 000000001337001fh
0x1337001f pop rbx
0x13370020 mov rax,2
0x13370027 mov rdi,rbx
0x1337002a mov rsi,0
0x13370031 mov rdx,0
0x13370038 syscall
0x1337003a mov rdi,rax
0x1337003d mov rax,0
0x13370044 mov rsi,rsp
0x13370047 mov rdx,96h
0x1337004e syscall

```

Before

Stack

```

000055f58cffd2b
00007ffe03778c08
0000000010000000
000000f48cffd2b0
0000000013370000
00007ffe03778c00
0000000000000000
000055f58cffd2b0

```

Registers

```

rax = 0x13370000
Data pointer: e81a0000002f7661...
rbx = 0x00000000
(nil)
rcx = 0x00000000
(nil)
rdx = 0x00000000
(nil)
rsi = 0x00000000
(nil)
rdi = 0x00000000
(nil)
rbp = 0x00000000
(nil)
rsp = 0x7ffe03778ad8
Data pointer: 8bd2ff8cf5550000...

```

After

Stack

```

0000000013370005
000055f58cffd2b
00007ffe03778c08
0000000020000000
000000f48cffd2b0
0000000013370000
00007ffe03778c00
0000000000000000

```

Registers

```

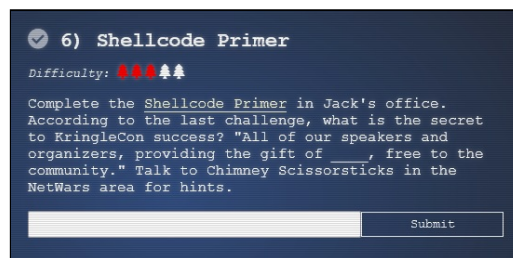
rax = 0x13370000
Data pointer: e81a0000002f7661...
rbx = 0x00000000
(nil)
rcx = 0x00000000
(nil)
rdx = 0x00000000
(nil)
rsi = 0x00000000
(nil)
rdi = 0x00000000
(nil)
rbp = 0x00000000
(nil)
rsp = 0x7ffe03778ad0
Data pointer: 0500371300000000...

```

8. Close the Debugger window

Stdout

Secret to KringleCon success: all of our speakers and organizers, providing the gift of cyber security knowledge, free to the community.

9. Close **Shellcode Primer**10. Click the **Tick** (Objectives) icon11. Click **6) Shellcode Primer**12. Type **cyber security knowledge**13. Click the **Submit** button