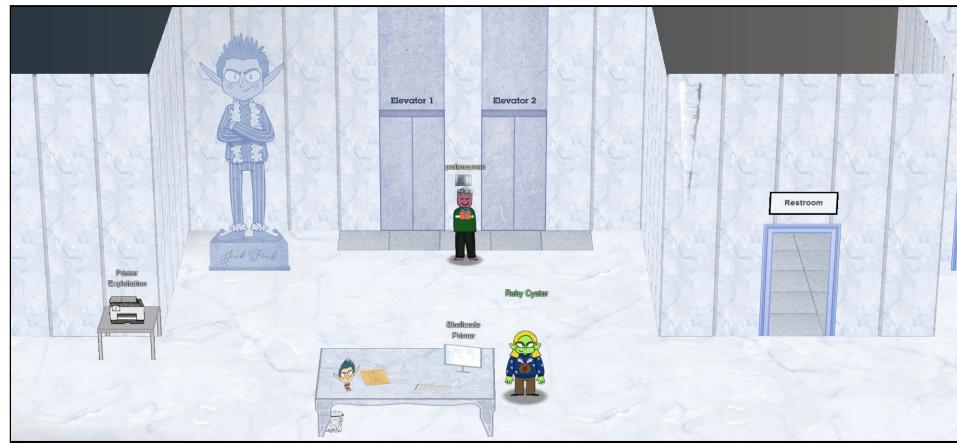


KringleCon 4: Calling Birds!

Jack's Office



1. Click to talk to Ruby Cyster

Hey, I'm Ruby Cyster. Don't listen to anything my sister, Ingreta, says about me.

So I'm looking at this system, and it has me a little bit worried.

If I didn't know better, I'd say someone here is learning how to hack North Pole systems.

Who's got that kind of nerve!

Anyway, I hear some elf on the other roof knows a bit about this type of thing.

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 <code>metasm</code> Ruby library to assemble, in case you want to work on your code locally: <pre>require 'metasm' assembled = Metasm::Shellcode.assemble(Metasm::X86_64.new, payload['code']).encode_string.unpack('H*').pop()</pre> 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. Click 1. Introduction

The goal of Shellcode Primer is to teach you how to write some basic x64 shellcode for reading a file. We'll take you through each piece of what you need, step by step, and show you what's going on.

First, let's learn the user interface a bit. There's some code below. The left is where you type code, and the right will attempt to syntax-highlight and show build errors.

For the time being, you don't need to change anything, just have a look at what it's doing - it's more or less the same type of stuff you're going to be learning.

Go ahead and execute the code (using the bottom below) and play around in the debugger. On the left, you'll see instructions executing in the order that they execute. Click on them to see the state when that instruction executes!

Also, don't forget to click that *hint* button below! Hints don't cost you anything.:)

```
; Set up some registers (sorta like variables) with values
; In the debugger, look how these change!
mov rax, 0
mov rbx, 1
mov rcx, 2
mov rdx, 3
mov rsi, 4
mov rdi, 5
mov rbp, 6

; Push and pop - watch how the stack changes!
push 0x12345678
pop rax

; Set up some registers (sorta like variables) with values
; In the debugger, look how these change!
mov rax, 0
mov rbx, 1
mov rcx, 2
mov rdx, 3
mov rsi, 4
mov rdi, 5
mov rbp, 6

; Push and pop - watch how the stack changes!
push 0x1111
push 0x2222
push 0x3333
pop rax
```

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

Reset **Execute** Assembles to: 48c7c0000000048c7c3010000048c7c1020000048c7c2030000048c7c6040000048c7c7050000048c7c506000006878563412586811100006822200068333000585858e00d0000048656c6c6f20576f72

4. Click the Request Hint [0/1] button

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

- Congratulations, you found the first hint! Others might be more helpful, so keep clicking (on other levels)!

5. Click the Execute button

	Before	After
Registers	rax = 0x13370000 Data pointer: 48c7c0000000048...	rax = 0x00000000 (nil)
Stack	00005f548b0328b rbx = 0x00000000 (nil) 00000020000000 rcx = 0x00000000 (nil) 00000c848b032b0 rdx = 0x00000000 (nil) 000000013370000 rsi = 0x00000000 (nil) 00007fff17afc340 rdx = 0x00000000 (nil) 0000000000000000 rsi = 0x00000000 (nil) 000055f548b032b0 rbp = 0x00000000 (nil) rsp = 0x7fff17afc218 Data pointer: 8b32b048f555000...	00005f548b0328b rbx = 0x00000000 (nil) 00000020000000 rcx = 0x00000000 (nil) 00000c848b032b0 rdx = 0x00000000 (nil) 000000013370000 rsi = 0x00000000 (nil) 00007fff17afc340 rdx = 0x00000000 (nil) 0000000000000000 rsi = 0x00000000 (nil) 000055f548b032b0 rbp = 0x00000000 (nil) rsp = 0x7fff17afc218 Data pointer: 8b32b048f555000...

6. Close the Debugger window

Congratulations! You completed Introduction! The next level has been unlocked

7. Click 2. Loops

Shellcode Primer	Loops
Home	Although you won't have to worry about writing a loop for any of these lessons, showing how a loop works is a good demo for the debugger.
1. Introduction ✓	Look at the code below, then execute it (no need to change it). Watch how the same code repeats, over and over, with <code>rax</code> changing in each loop.
2. Loops	Notice how the code listing below isn't the same as what is executed in the debugger. In the History section of the debugger, the instructions will change to show what is executed to achieve what you describe in the assembly source code. <div style="border: 1px solid #ccc; padding: 10px;"><pre>; We want to loop 5 times - you can change this if you want! mov rax, 5 ; Top of the loop top: ; Decrement rax dec rax ; Jump back to the top until rax is zero jnz top ; Cleanly return after the loop ret</pre></div> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><pre>; We want to loop 5 times - you can change this if you want! mov rax, 5 ; Top of the loop top: ; Decrement rax dec rax ; Jump back to the top until rax is zero jnz top ; Cleanly return after the loop ret</pre></div> <p>No Hints - Hints are free!</p> <p>Reset Execute Assembles to: 48c7c00500000048ff875fb3</p>

8. Click the **Execute** button

Registers	Registers
rax = 0x13370000	rax = 0x00000005
Data pointer: 48c7c00500000045...	(not a pointer)
Stack	Stack
0000560f949c12b8 00007ffdc8d65b68 (nil) 0000000200000000	0000560f949c12b8 00007ffdc8d65b68 (nil) 0000000200000000
Registers	Registers
rcx = 0x00000000	rcx = 0x00000000
0000001a949c12b0 0000000813370000 (nil)	0000001a949c12b0 0000000813370000 (nil)
rdx = 0x00000000	rdx = 0x00000000
00007ffdcd8d65b60 0000000000000000 (nil)	0000000000000000 (nil)
rsi = 0x00000000 (nil)	rsi = 0x00000000 (nil)
rdi = 0x00000000 (nil)	rdi = 0x00000000 (nil)
rbp = 0x00000000 (nil)	rbp = 0x00000000 (nil)
rsp = 0x7ffdc8d65a38 Data pointer: 8b129c940f560000...	rsp = 0x7ffdc8d65a38 Data pointer: 8b129c940f560000...
Registers	Registers
exit code	Success!
Process exited cleanly with exit code 0	Let's try writing some code now!
History	
0x13370000 mov rax,5	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 dec rax	
0x13370008 jne short 0000000013370007h	
0x13370007 ret	

9. Close the Debugger window

Let's try writing some code now!

10. Click 3. Getting Started

Shellcode Primer

Home
1. Introduction ✓
2. Loops ✓
3. Getting Started

Getting Started

Welcome! Are you ready to learn how to write shellcode? We hope so! First, some tips:

- Comments are denoted with a semicolon (;)
- Don't forget to look at the debugger, line by line, if something is wrong
- Really, don't forget to read the error list!** We check each place where you might go wrong in your code
- Your code for each level is saved in your browser, so you can leave and come back, refresh the page, and hop back to previous levels to borrow code

This level currently fails to build because it has no code. Can you add a `return` statement at the end? Don't worry about what it's actually returning (yet)!

Feel free to check previous levels!

```
; This is a comment! We'll use comments to help guide your journey.  
; Right now, we just need to RETurn!  
;  
; Enter a return statement below and hit Execute to see what happens!
```

```
; This is a comment! We'll use comments to help guide your journey.  
; Right now, we just need to RETURN!  
  
; Enter a return statement below and hit Execute to see what happens!
```

[Request Hint \[0 / 2\]](#) - Hints are free!

[Reset](#) [Execute](#) [No code!](#)

11. Click the Request Hint [0/2] button and then click the Request Hint [1/2] button



12. Modify the contents of the code window to

```
; This is a comment! We'll use comments to help guide your journey.  

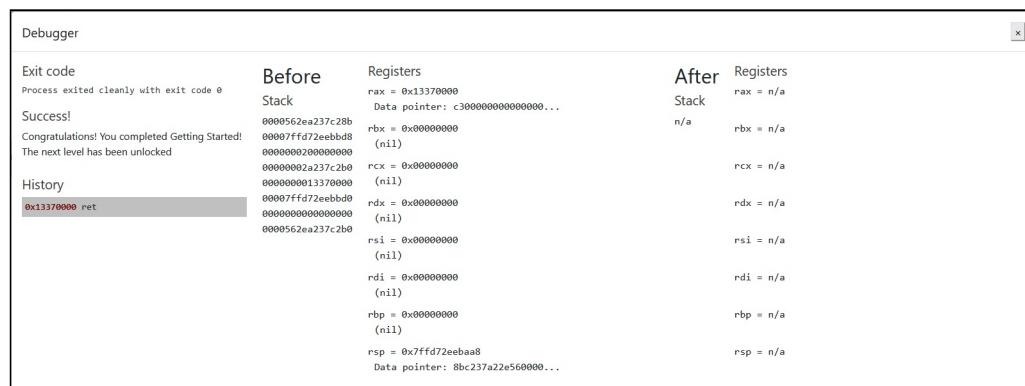
; Right now, we just need to RETurn!  

;  

; Enter a return statement below and hit Execute to see what happens!  

ret
```

13. Click the Execute button



14. Close the Debugger window

Congratulations! You completed Getting Started! The next level has been unlocked

15. Click 4. Returning a Value

The page includes a sidebar with navigation links: Home, 1. Introduction (checked), 2. Loops (checked), 3. Getting Started (checked), and 4. Returning a Value (checked). The main content area is titled "Returning a Value" and contains the following text:

Now that we have an empty function, we can start building some code! Let's learn what a register is.

A register is like a variable, except there are a small number of them - you have about eight general purpose 64-bit integers registers on amd64 (we won't talk about floating point or other special registers):

- rax
- rbx
- rcx
- rdx
- rdi
- rsi
- rbp
- rsp

All mathy stuff that a computer does (add, subtract, xor, etc) operates on registers, not directly on memory. So they're super important!

Specific registers have some implicit meaning, mostly by convention. For example, when a function returns, its return value is typically put in `rax`. Let's do that!

To move a value into a register, use the `mov` instruction; for example:

```
mov rdx, 1
```

In a higher-level language this would be equivalent to:

```
rdx = 1 --
```

For this level, can you return the number '1337' from your function?

That means that `rax` must equal `1337` when the function returns.

```
: TODO: Set rax to 1337
```

```
: Return, just like we did last time
ret
```

```
; TODO: Set rax to 1337

; Return, just like we did last time
ret
```

At the bottom of the page are buttons for "Request Hint [0 / 4]" and "Execute". Below the "Execute" button is a note: "Assembles to: C".

16. Click the Request Hint [0/4] button and then click the Request Hint [1/4] button

17. Click the Request Hint [2/4] button and then click the Request Hint [3/4] button

Request Hint [4 / 4]

- Hints are free!

- Unlike most languages, you don't return values with `ret x`; instead, you set `rax` to a value, then return
- To set a register, use `mov REGNAME, value`; for example, `mov rcx, 123`
- The return register is `rax`
- If something isn't working, don't forget to check the debugger that pops up when you click Execute! It'll show you the state of every register at every step

18. Modify the contents of the code window to

```

; TODO: Set rax to 1337
mov rax, 1337

; Return, just like we did last time
ret

```

19. Click the Execute button

Debugger

Exit code Process exited cleanly with exit code 57	Before Registers <code>rax = 0x13370000</code> <code>Data pointer: 48c7c039050000c3...</code>	After Registers <code>rax = 0x00000539</code> <code>(not a pointer)</code>
Success! Congratulations! You completed Returning a Value! The next level has been unlocked	Stack <code>000055db5f34528b</code> <code>rbx = 0x00000000</code> <code>(nil)</code>	<code>000055db5f34528b</code> <code>rbx = 0x00000000</code> <code>(nil)</code>
History <code>0x13370000 mov rax,539h</code> <code>0x13370007 ret</code>	<code>00000000105f3452b0</code> <code>rcx = 0x00000000</code> <code>(nil)</code>	<code>00000000105f3452b0</code> <code>rcx = 0x00000000</code> <code>(nil)</code>
	<code>00000000013370000</code> <code>rdx = 0x00000000</code> <code>(nil)</code>	<code>00000000013370000</code> <code>rdx = 0x00000000</code> <code>(nil)</code>
	<code>0000000000000000</code> <code>rsi = 0x00000000</code> <code>(nil)</code>	<code>0000000000000000</code> <code>rsi = 0x00000000</code> <code>(nil)</code>
	<code>0000000000000000</code> <code>rdi = 0x00000000</code> <code>(nil)</code>	<code>0000000000000000</code> <code>rdi = 0x00000000</code> <code>(nil)</code>
	<code>0000000000000000</code> <code>rbp = 0x00000000</code> <code>(nil)</code>	<code>0000000000000000</code> <code>rbp = 0x00000000</code> <code>(nil)</code>
	<code>0000000000000000</code> <code>rsp = 0x7ffffb3eb76d8</code> <code>Data pointer: 8b52345fdb550000...</code>	<code>0000000000000000</code> <code>rsp = 0x7ffffb3eb76d8</code> <code>Data pointer: 8b52345fdb550000...</code>

20. Close the Debugger window

Congratulations! You completed Returning a Value! The next level has been unlocked

21. Click 5. System Calls

The screenshot shows the 'System Calls' challenge in the Shellcode Primer. The challenge description asks to find the syscall number for `sys_exit` and put it in `rax`, then move the exit code (99) into `rdi`, and finally perform the syscall. The code window has a placeholder assembly script. At the bottom left, there is a blue button labeled 'Request Hint [0 / 4] - Hints are free!'. Below the code window, there are three buttons: 'Start', 'Execute', and 'Assembles to: 0000'.

22. Click the Request Hint [0/4] button and then click the Request Hint [1/4] button

23. Click the Request Hint [2/4] button and then click the Request Hint [3/4] button

The screenshot shows the 'Request Hint [4 / 4]' step. It contains a list of hints: 'Start by consulting a syscall table and finding the number for `sys_exit`, and moving that into `rax`', 'Next, `mov` the desired return value into `rdi` (the first parameter to every syscall)', 'Finally, use the literal `syscall` instruction', and 'If something isn't working, be sure to read the problem list in the debugger! It tries to detect any and all issues'.

24. Modify the contents of the code window to

```
; TODO: Find the syscall number for sys_exit and put it in rax
mov rax, 60
; TODO: Put the exit_code we want (99) in rdi
mov rdi, 99
; Perform the actual syscall
syscall
```

25. Click the Execute button

Debugger	
Exit code	
Process exited cleanly with exit code 99	
Success!	
Great job! We'll come back to syscalls in a bit, but let's look at something completely different in the next exercise.	
History	
<code>0x13370000 mov rax,3Ch</code>	
<code>0x13370007 mov rdi,63h</code>	
<code>0x1337000e syscall</code>	

Before	Registers	After	Registers
Stack	<code>rax = 0x13370000</code> <code>Data pointer: 48c7c03c00000048...</code>	Stack	<code>rax = 0x0000003c</code> <code>(not a pointer)</code>
000055a6dadf028b	<code>rbx = 0x00000000</code> <code>00007ffe087d9818 (nil)</code>	000055a6dadf028b	<code>rbx = 0x00000000</code> <code>00007ffe087d9818 (nil)</code>
0000000020000000	<code>rcx = 0x00000000</code> <code>0000000020dadf02b0 (nil)</code>	0000000020000000	<code>rcx = 0x00000000</code> <code>0000000020dadf02b0 (nil)</code>
0000000000000000	<code>rdx = 0x00000000</code> <code>0000000000000000 (nil)</code>	0000000000000000	<code>rdx = 0x00000000</code> <code>0000000000000000 (nil)</code>
0000000013370000	<code>rsi = 0x00000000</code> <code>0000000013370000 (nil)</code>	0000000013370000	<code>rsi = 0x00000000</code> <code>(nil)</code>
00007ffe087d9810	<code>rdi = 0x00000000</code> <code>0000000000000000 (nil)</code>	00007ffe087d9810	<code>rdi = 0x00000000</code> <code>(nil)</code>
0000000000000000	<code>rbp = 0x00000000</code> <code>0000000000000000 (nil)</code>	0000000000000000	<code>rbp = 0x00000000</code> <code>(nil)</code>
000055a6dadf02b0	<code>rsp = 0x7ffe087d96e8</code> <code>Data pointer: 8b02dfdaa6550000...</code>	000055a6dadf02b0	<code>rsp = 0x7ffe087d96e8</code> <code>Data pointer: 8b02dfdaa6550000...</code>

26. Close the Debugger window

Great job! We'll come back to syscalls in a bit, but let's look at something completely different in the next exercise.

27. Click 6. Call Into the Void

Shellcode Primer

Calling Into the Void

Before we learn how to use the `Really Good` syscalls, let's try something fun: crash our shellcode on purpose!

You might think I'm mad, but there's a method to my madness. Run the code below and watch what happens! No need to modify it, unless you want to.:-)

Be sure to look at the debugger to see what's going on! Especially notice the top of the stack at the `ret` instruction.

```
; Push this value to the stack
push 0x12345678

; Try to return
ret
```

Request Hint [0 / 2] - Hints are free!

Reset **Execute** Assembles to: 6878563412c3000...

28. Click the Request Hint [0/2] button and then click the Request Hint [1/2] button

Request Hint [2 / 2] - Hints are free!

- This challenge is already solved - just click **Execute** and win! ...but...
- It's valuable to look at the debugger to help understand what's going on - how did we end up executing code at 0x12345678?

29. Click the Execute button

Debugger

Exit code
Execution crashed with a segmentation fault
(SIGSEGV) @ 0x12345678

Success!
There, did you see? It crashed when trying to execute code at 0x12345678! Meet me in the next level, and we'll see what's going on!

History

```
0x13370000 push 12345678h
0x13370005 ret
0x12345678 <invalid memory>
```

Before	Registers	After	Registers
Stack	rax = 0x13370000 Data pointer: 6878563412c30000...	Stack	rax = 0x13370000 Data pointer: 6878563412c30000...
000055ec90e2c2b8	rbx = 0x00000000 (nil)	0000000012345678	rbx = 0x00000000 (nil)
00007fff56d42488	rcx = 0x00000000 (nil)	0000055ec90e2c2b8	rcx = 0x00000000 (nil)
0000000020000000	rdx = 0x00000000 (nil)	0000000020000000	rdx = 0x00000000 (nil)
0000000090e2c2b8	rsi = 0x00000000 (nil)	00000000c90e2c2b8	rsi = 0x00000000 (nil)
0000000013370000	rdi = 0x00000000 (nil)	0000000013370000	rdi = 0x00000000 (nil)
000007fff56d42480	rbp = 0x00000000 (nil)	000000007fff56d42480	rbp = 0x00000000 (nil)
0000000000000000	rsp = 0x7ffff56d42358 Data pointer: 8bc2e290ec550000...	0000000000000000	rsp = 0x7ffff56d42350 Data pointer: 7856341200000000...

30. Close the Debugger window

There, did you see? It crashed when trying to execute code at 0x12345678! Meet me in the next level, and we'll see what's going on!

31. Click 7. Getting RIP

Shellcode Primer

Getting RIP

What happened in the last exercise? Why did it crash at `call [eax]`? And did you notice that 0x12345678 was on top of the stack when `ret` happened?

The short story is this: `call` pushes the return address onto the stack, and `ret` jumps to it. Whaaaaaa!

This is going to be long, but hopefully will make it all clear!

Let's back up a bit. At any given point, the instruction currently being executed is stored in a special register called the instruction pointer (`rip`), which you may also hear called a program counter (`pc`).

What is the `rip` value at the first line in our code? Well, since we have a debugger, we know that it's `0x13370000`. But sometimes you don't know and need to find out.

The most obvious answer is to treat it like a normal register, like this:

```
mov rip, rip
ret
```

Does that work? Nope! You can't directly access `rip`. That means we need a trick.

When you use `push rip`, the CPU doesn't care where it's calling, or whether there's a `---` waiting for it. The CPU assumes that, if the author put a `rip` in, there will naturally be a `ret` on the other end. Doing anything else would just be silly! So `push` pushes the return address onto the stack before jumping into a function. When the function complete, the `ret` instruction uses the return address on the stack to know where to return to.

The CPU assumes that, sometime later, a `ret` will execute. The `ret` assumes that at some point earlier a `push` happened, and that means that the top of the stack has the return address. The `ret` will retrieve the return address off the top of the stack (using `pop`) and jump to it.

Of course, we can execute `pop` too! If we `pop` the return address off the stack, instead of jumping to it, the address goes into a register. Hmml! Does that also sound like `rip`, `rip` to you?

For this exercise, can you `pop` the address after the call - the `0f 34 [rip]` instruction - into `rip` then return?

```
; Remember, this call pushes the return address to the stack
call place_below_the_nop

; This is where the function "thinks" it is supposed to return
nop

; This is a "label" - as far as the call knows, this is the start of a function
place_below_the_nop:

; TODO: Pop the top of the stack into rip

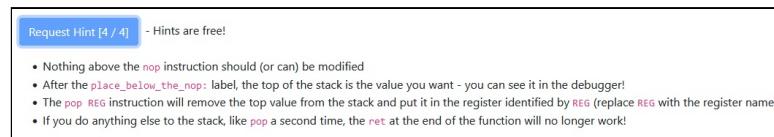
; Return from our code, as in previous levels
ret
```

Request Hint [0 / 4] - Hints are free!

Reset **Execute** Assembles to: e80100000090c3

32. Click the Request Hint [0/4] button and then click the Request Hint [1/4] button

33. Click the Request Hint [2/4] button and then click the Request Hint [3/4] button



34. Modify the contents of the code window to

```
; Remember, this call pushes the return address to the stack
call place_below_the_nop

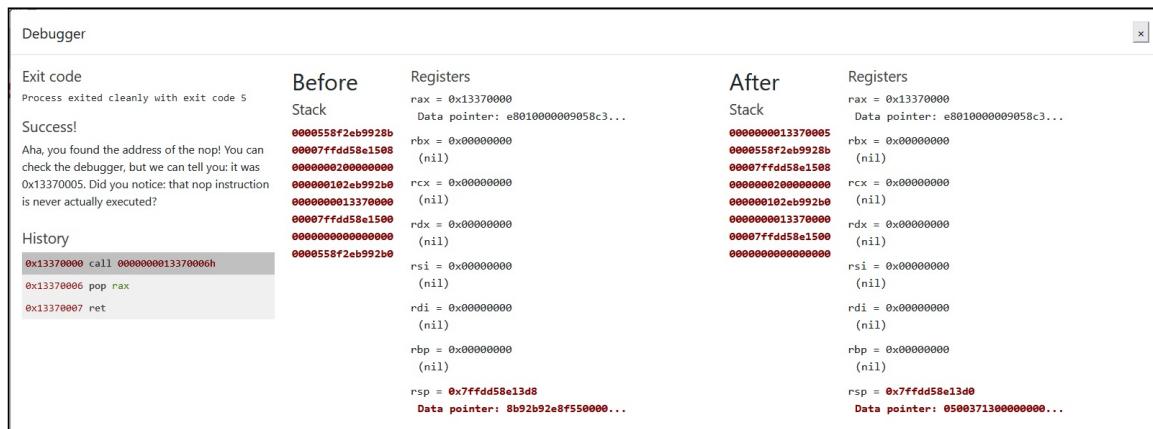
; This is where the function *thinks* it is supposed to return
nop

; This is a 'label' - as far as the call knows, this is the start of a function
place_below_the_nop:

; TODO: Pop the top of the stack into rax
pop rax

; Return from our code, as in previous levels
ret
```

35. Click the Execute button



36. Close the Debugger window

Aha, you found the address of the nop! You can check the debugger, but we can tell you: it was 0x13370005. Did you notice: that nop instruction is never actually executed?

37. Click 8. Hello, World!

Shellcode Primer

Hello, World!

So remember how last level, we got the address of `nop` and returned it?

Did you see that `nop` execute? Nope! We jumped right over it, but stored its address en-route. What can we do by knowing our own address?

Well, since shellcode is, by definition, self-contained, you can do other fun stuff like include data alongside the code!

What if the return address isn't an instruction at all, but a string?

For this next exercise, we include a plaintext string - 'Hello World' - as part of the code. It's just sitting there in memory. If you look at the compiled code, it's all basically `Hello World`, which doesn't run.

Instead of trying to run it, can you `call` past it, and `pop` its address into `rax`?

Don't forget to check the debugger after to see it in `rax`.

```
; This would be a good place for a call
; This is the literal string 'Hello World', null terminated, as code. Except
; it'll crash if it actually tries to run, so we'd better jump over it!
db 'Hello World',0

; This would be a good place for a label and a pop

; This would be a good place for a re... oh wait, it's already here. Hooray!
ret
```

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

Reset Execute Assembles to: 48656c6c6f20576f726c6400c3

38. Click the Request Hint [0/1] button

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

- This is pretty much the same as the previous challenge, except `nop` has been replaced with a string

39. Modify the contents of the code window to

```
; This would be a good place for a call
call str_ref
; This is the literal string 'Hello World', null terminated, as code. Except
; it'll crash if it actually tries to run, so we'd better jump over it!
db 'Hello World',0

; This would be a good place for a label and a pop
str_ref:
pop rax
; This would be a good place for a re... oh wait, it's already here. Hooray!
ret
```

40. Click the Execute button

Registers	
Before	After
Stack	Stack
rax = 0x13370000 Data pointer: e80c00000048656c...	rax = 0x13370000 Data pointer: e80c00000048656c...
rbx = 0x00000000	rbx = 0x00000000
(nil)	(nil)
rcx = 0x00000000	rcx = 0x00000000
rdx = 0x00000000	rdx = 0x00000000
(nil)	(nil)
r8 = 0x00000000	r8 = 0x00000000
(nil)	(nil)
r9 = 0x00000000	r9 = 0x00000000
(nil)	(nil)
rsi = 0x00000000	rsi = 0x00000000
(nil)	(nil)
rdi = 0x00000000	rdi = 0x00000000
(nil)	(nil)
rbp = 0x00000000	rbp = 0x00000000
(nil)	(nil)
rsp = 0x7ffd6b0dc528	rsp = 0x7ffd6b0dc528
Data pointer: 8bb2c7da25560000...	Data pointer: 0500371300000000...

41. Close the Debugger window

Hello World, indeed! Congratulations, you've unlocked the next exercise.

42. Click 9. Hello, World!!

Shellcode Primer

Hello, World!!

Remember syscalls? Earlier, we used them to call an exit. Now let's try another!

This time, instead of getting a pointer to the string `Hello World`, we're going to print it to standard output (stdout).

Have another look at [the syscall table](#). Can you find `sys_write`, and use to print the string `Hello World!` to stdout?

Note: stdout's file descriptor is 1.

```

; TODO: Get a reference to this string into the correct register
db 'Hello World!',0

; Set up a call to sys_write
; TODO: Set rax to the correct syscall number for sys_write

; TODO: Set rdi to the first argument (the file descriptor, 1)

; TODO: Set rsi to the second argument (buf - this is the "Hello World" string)

```

Request Hint [0 / 4] - Hints are free!

Reset Execute Assembles to: 48656c6c6f20576f726c6421000f05c

43. Click the Request Hint [0/4] button and then click the Request Hint [1/4] button

44. Click the Request Hint [2/4] button and then click the Request Hint [3/4] button

Request Hint [4 / 4] - Hints are free!

- This is a combination of the last several levels
- Refer to [System Calls](#), but use `sys_write` instead of `sys_exit` and update the arguments accordingly
- Refer to [Hello, World!](#) on how to get a string reference into a register (i.e., `call/pop`)
- Don't forget to read the boilerplate comments - they indicate which registers to use

45. Modify the contents of the code window to

```

; TODO: Get a reference to this string into the correct register
call str_ref
db 'Hello World!',0
str_ref:
pop rbx
; Set up a call to sys_write
; TODO: Set rax to the correct syscall number for sys_write
mov rax, 1

; TODO: Set rdi to the first argument (the file descriptor, 1)
mov rdi, 1

; TODO: Set rsi to the second argument (buf - this is the "Hello World" string)
mov rsi, rbx

; TODO: Set rdx to the third argument (length of the string, in bytes)
mov rdx, 12

; Perform the syscall
syscall

; Return cleanly
ret

```

46. Click the **Execute** button

The screenshot shows the debugger interface with two columns: 'Before' and 'After'. The 'Before' column displays assembly instructions and their corresponding register values. The 'After' column shows the state after execution, with updated register values. The assembly code includes calls to `sys_open` and `sys_write`, and a `syscall` instruction.

```

Registers Before
rax = 0x13370000
Data pointer: e80d00000048656c...
Stack
0000556ae481e28b rbx = 0x00000000
00007ffe37e91408 (nil)
0000000200000000
00000005ce481e2b0 rcx = 0x00000000
0000000013370000 (nil)
00000007ffe37e91408 rdx = 0x00000000
0000000000000000 (nil)
0000556ae481e2b0 rsi = 0x00000000
(nil)
0x1337000012 pop rbx
0x13370013 mov rax,1
0x1337001a mov rdi,1
0x13370021 mov rs1,rbx
0x13370024 mov rdx,0Ch
0x1337002b syscall
0x1337002d ret

Registers After
rax = 0x13370000
Data pointer: e80d00000048656c...
Stack
0000000013370005 rbx = 0x00000000
0000556ae481e2b0 (nil)
00007ffe37e91408 rcx = 0x00000000
0000000200000000 (nil)
00000005ce481e2b0 rdx = 0x00000000
0000000013370000 (nil)
00000007ffe37e91408 rs1 = 0x00000000
(nil)
0x13370012 pop rdi
0x13370013 mov rbp,0x00000000
0x1337001a mov rbp,0x00000000
0x13370021 mov rbp,0x00000000
0x13370024 mov rsp,0x7ffe37e912d8
Data pointer: 8be281e46a550000...

```

47. Close the Debugger window

Congratulations! You completed Hello, World!!! The next level has been unlocked

48. Click 10. Opening a File

The screenshot shows the 'Shellcode Primer' interface with a sidebar of challenges and the main area displaying assembly code for opening a file. The assembly code includes comments for each step of the process: getting a reference to the string, setting up a call to `sys_open`, setting arguments, performing the syscall, and returning the file handle. A 'Request Hint [0 / 2]' button is visible at the bottom left.

```

; TODO: Get a reference to this string into the correct register
db '/etc/passwd',0

; Set up a call to sys_open
; TODO: Set rax to the correct syscall number

; TODO: Set rdi to the first argument (the filename)

; TODO: Set rsi to the second argument (flags - 0 is fine)

; TODO: Set rdx to the third argument (mode - 0 is also fine)

; Perform the syscall
syscall

; syscall sets rax to the file handle, so to return the file handle we don't
; need to do anything else!
ret

```

49. Click the **Request Hint [0/2]** button and then click the **Request Hint [1/2]** button

Request Hint [2 / 2] - Hints are free!

- This is nearly the same as `Hello, World!!`, only using `sys_open` instead of `sys_write`!
- Don't forget to read the boilerplate comments!

50. Modify the contents of the code window to

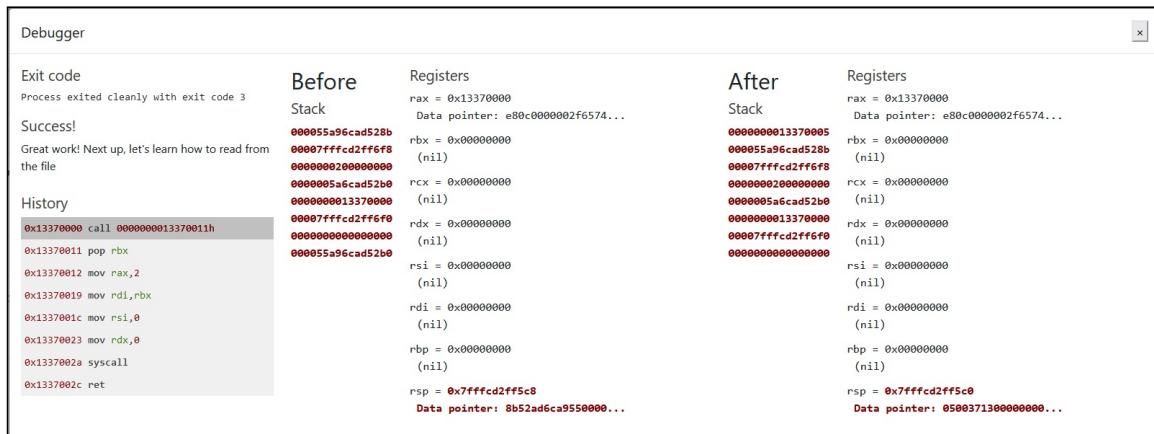
```

; TODO: Get a reference to this string into the correct register
call str_ref
db '/etc/passwd',0
str_ref:
pop rbx
; Set up a call to sys_open
; TODO: Set rax to the correct syscall number
mov rax, 2
; TODO: Set rdi to the first argument (the filename)
mov rdi, rbx
; TODO: Set rsi to the second argument (flags - 0 is fine)
mov rsi, 0
; TODO: Set rdx to the third argument (mode - 0 is also fine)
mov rdx, 0
; Perform the syscall
syscall

; syscall sets rax to the file handle, so to return the file handle we don't
; need to do anything else!
ret

```

51. Click the **Execute** button



52. Close the Debugger window

Great work! Next up, let's learn how to read from the file

53. Click 11. Reading a File

Shellcode Primer

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. You can use `rax` for `count` - temporary memory - `rax` 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.

Finally, if you use `rax` 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 `rax` 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: 2f7661722feef727466706fc657365053725574732e74797400

54. Click the Request Hint [0/1] button

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

- This is the last challenge. Good luck!

55. 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
```

56. Click the Execute button

The screenshot shows a debugger interface with two columns: 'Before' and 'After'. The 'Before' column displays assembly code and register values. The 'After' column shows the state after execution. A grey box highlights the assembly code:

```

    Debugger

    Exit code
    Process exited cleanly with exit code 99

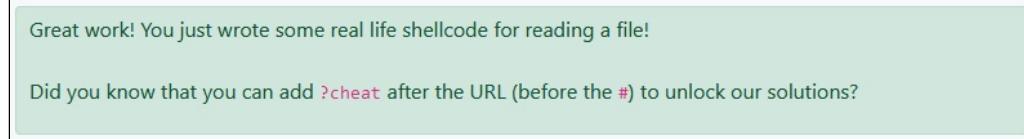
    Stdout
    Secret to KringleCon success: all of our speakers and organizers, providing the gift of cyber security knowledge, free to the community.

    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 0000000001337001Fh
    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
    0x13370050 mov rax,1
    0x13370057 mov rdi,1
    0x1337005e mov rsi,rsp
    0x13370061 mov rdx,96h
    0x13370068 syscall
    0x1337006a mov rax,3Ch
    0x13370071 mov rdi,63h
    0x13370078 syscall
  
```

57. Close the Debugger window

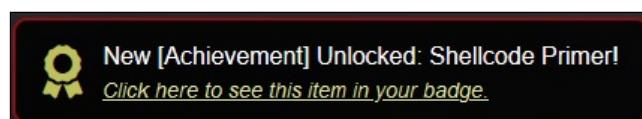
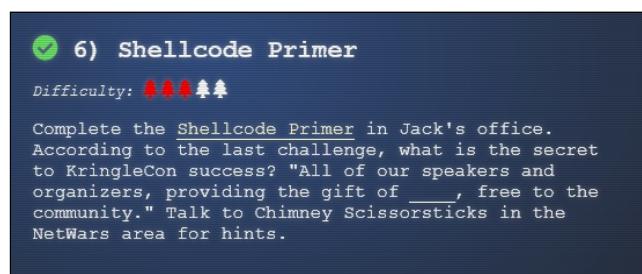


58. Close the Shellcode Primer and then click the Tick (Objectives) icon

59. Click 6) Shellcode Primer

60. Type cyber security knowledge

61. Click the Submit button

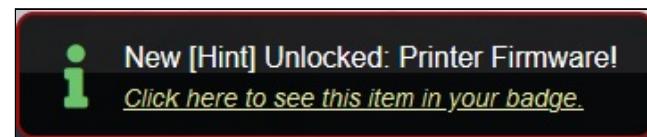


62. Click to talk to Ruby Cyster

Oh man - what is this all about? Great work though.

So first things first, you should definitely take a look at the firmware.

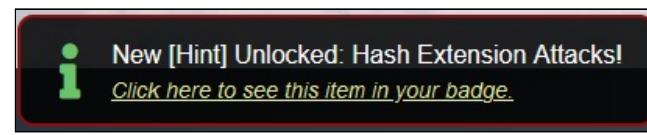
With that in-hand, you can pick it apart and see what's there.



63. Click to talk to Ruby Cyster

Did you know that if you append multiple files of that type, the last one is processed?

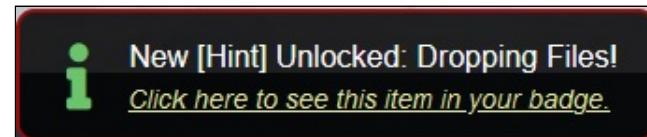
Have you heard of Hash Extension Attacks?



64. Click to talk to Ruby Cyster

If something isn't working, be sure to check the output! The error messages are very verbose.

Everything else accomplished, you just might be able to get shell access to that dusty old thing!



65. Click the (Hints) icon

66. Click **Dropping Files**

A dark green rectangular card with white text. At the top, it says "Dropping Files". Below that, "From: Ruby Cyster" and "Objective: 7) Printer Exploitation". A horizontal dashed line separates this from the main message. The message reads: "Files placed in /app/lib/public/incoming will be accessible under https://printer.kringlecastle.com/incoming/." There is a dotted line at the bottom.

67. Click Hash Extension Attacks

Hash Extension Attacks

*From: Ruby Cyster
Objective: 7) Printer Exploitation*

Hash Extension Attacks can be super handy when there's some type of validation to be circumvented.

68. Click Printer Firmware

Printer Firmware

*From: Ruby Cyster
Objective: 7) Printer Exploitation*

When analyzing a device, it's always a good idea to pick apart the firmware. Sometimes these things come down Base64-encoded.

69. Click [Exit]

70. Click Printer Exploitation

The screenshot shows a web application for managing a printer. At the top, there are two printer icons. The left one has a yellow status bar above it that says "HOHOHOHOHOHOHOH! Cartridge very low". Below the printer icons is a sidebar with links: Status, Settings, Reports, and Firmware Update. The main content area is titled "Device Status - Refresh". It shows "Toner Status: Black Cartridge ~1%" with a progress bar indicating "Very Low, 161 estimated pages remain". Below this, there is a table for "Paper Input Tray" showing four trays with their status, capacity, size, and type. The first three trays are marked as "Low" (yellow), while the fourth is marked as "OK" (green). The "Paper Output Bin" section shows one bin with an "OK" status and a capacity of 550. At the bottom, there are details about the "Device Type: Monochrome Laser", "Speed: Up to 55 Pages/Minute", "Toner Cartridge Capacity: Approximately 25,000 Pages at 5% coverage", and "Maintenance Kit Life Remaining: 75%".

Paper Input Tray:	Status:	Capacity:	Size:	Type:
Tray 1	Low	550	Letter	Plain Paper
Tray 2	Low	550	Letter	Custom Type 2
Tray 3	OK	550	Legal	Custom Type 3
Tray 4	Low	550	Letter	Plain Paper

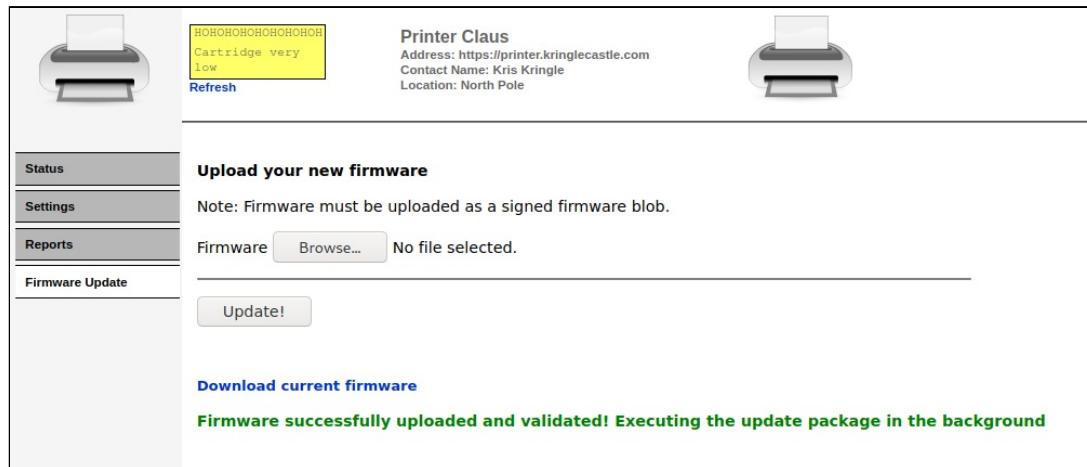
Paper Output Bin:	Status:	Capacity:
Standard Bin	OK	550

72. Click Firmware Update

73. Click the Browse... button

74. Select the newfirmware.json file

75. Click the **Update!** button

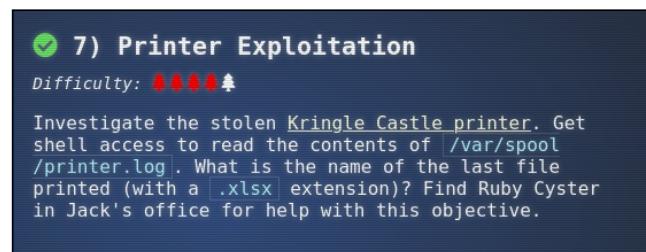


76. Click the **tick** (Objectives) icon

77. Click **7) Printer Exploitation**

78. Type **Troll_Pay_Chart.xlsx**

79. Click the **Submit** button



80. Click **[Exit]**

81. Move **Left** to enter the **Jack's Studio**