

软件漏洞： 内存损坏

分类漏洞

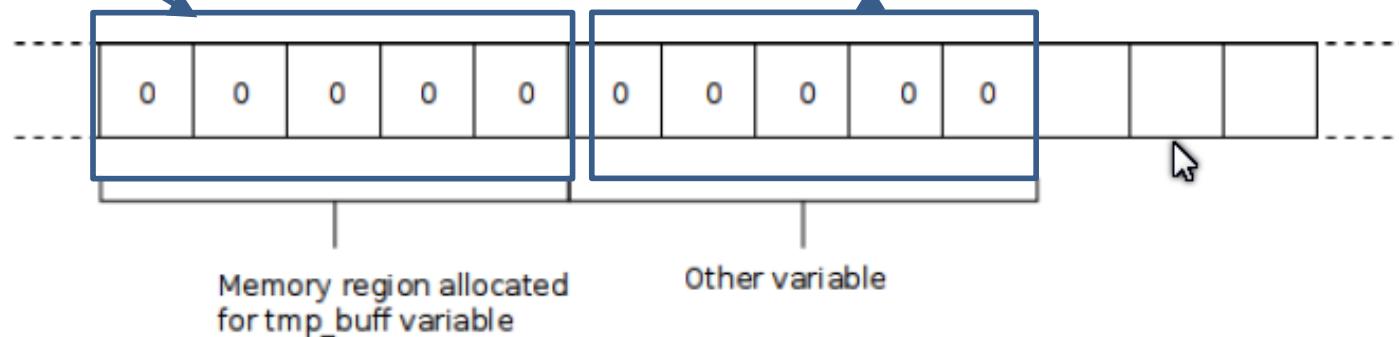
- 内存损坏 (安全)
 - 缓冲区溢出
 - 整数溢出
 - 堆栈溢出
 - 免费/双次使用
 - 空指针取消引用
- 利用内存损坏来控制程序
 - 格式化字符串攻击
 - 堆栈缓冲区溢出 (或基于堆栈的缓冲区溢出)
 - 堆缓冲区溢出 (或基于堆的缓冲区溢出)
 - 非“控制数据”存储器上的溢出
- 导致的攻击
 - 堆栈上的代码注入 (例如 , shellcode)
 - 代码重用技术 (返回libc , 面向返回的编程 , 面向跳转的编程)

缓冲区溢出

- 每当程序尝试将数据存储到缓冲区边界之外时，缓冲区溢出都会覆盖相邻的内存位置
- 源自编写代码时的错误
 - 不熟悉语言
 - 边界或算术错误
- 主要是C / C ++程序
- 通过具有自动内存管理的语言解决：动态范围检查（例如Java）或自动调整缓冲区大小（例如Perl）
 - 仍然是用C编写的本机库（例如JNI）

缓冲区溢出：基础知识

```
char src [] =“ ABCDEFGHI” ;  
字符tmp_buff [5];  
int password_checked;  
  
strcpy ( tmp_buff , src ) ;
```



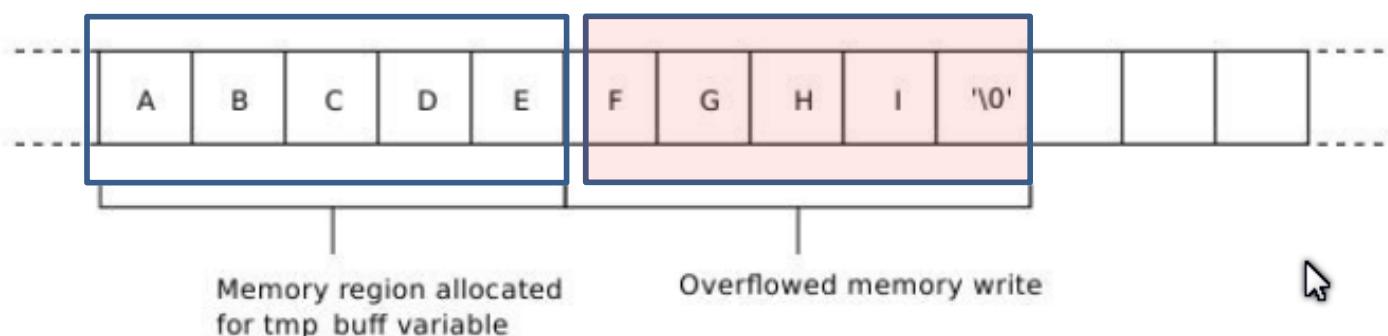
缓冲区溢出：在strcpy()之后

```
char src [] =“ ABCDEFGHI” ;
```

```
字符tmp_buff [5];
```

```
int password_checked;
```

```
strcpy ( tmp_buff , src ) ;
```



缓冲区溢出：通常的怀疑

- 不能正确检查字符串长度的字符串操作函数
 - gets() , strcpy() 等
- 复制参数不正确
 - memcpy() ...
- 所需内存长度的计算不正确
 - 零大小的malloc() s...

防止缓冲区溢出

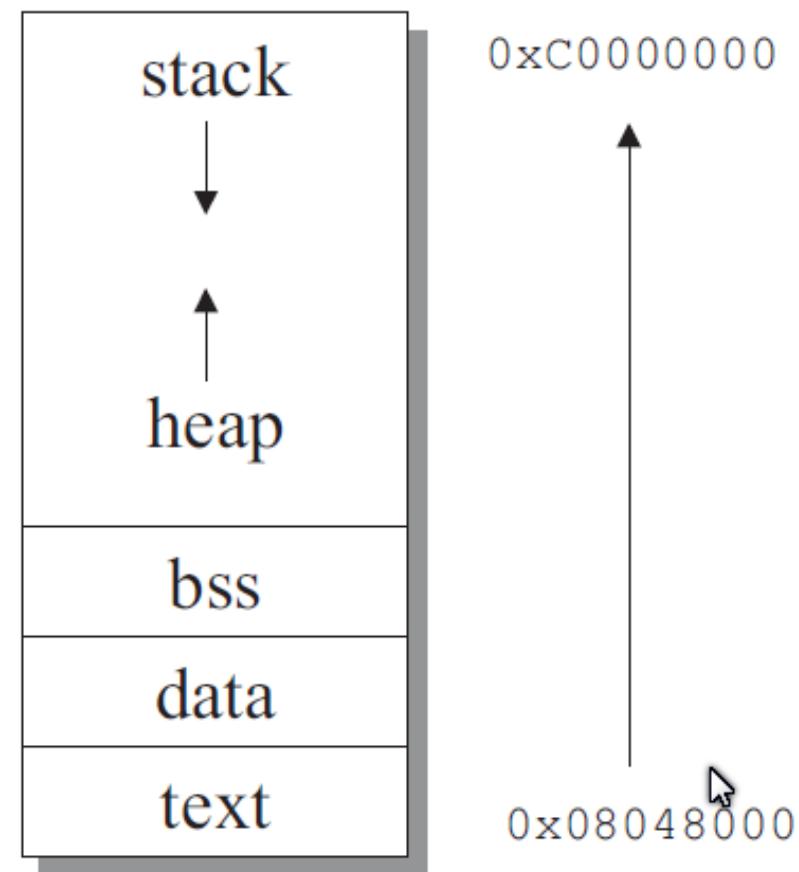
- 使用libc方法安全使用复制原语
 - 例如，限制要复制的数据量
 - `strncpy(tmp_buff , src , sizeof(tmp_buf))`
- 编译器扩展（如果可以计算数组长度）
- 注释（例如，副手，已完成...）

整数溢出

- 无符号整数
 - 32位：范围从0到 $2^{32}-1$ (4,294,967,295)
 - 溢出： $4,294,967,295 + 1 = 0$
- 签名整数
 - 32位：范围从 $- (2^{31})$ 到 $2^{31} - 1$ (2,147,483,647)
 - 溢出： $2,147,483,647 + 2 = -2,147,483,648$
- 某些语言 (Java , Ada) 会引发异常，而许多则不会
- 攻击者可以提供用于：
 - 计算缓冲区分配的大小 (malloc)
 - 数组访问 (特别是绑定检查)
- 解决方案：始终在关键操作之前检查溢出 !

有关内存布局的更多信息

- 文本
 - 也称为代码段全局初始化数据
- 数据
 -
- BSS
 - 全局未初始化数据
- 叠放
 - 局部变量
 - 也用于存储功能环境和参数
 - 通话期间（堆栈帧）
 - 生命力
 - 多线程：多栈
- 堆
 - 动态分配的变量
 - 通过calloc（）和malloc（）保留



程序存储器堆栈

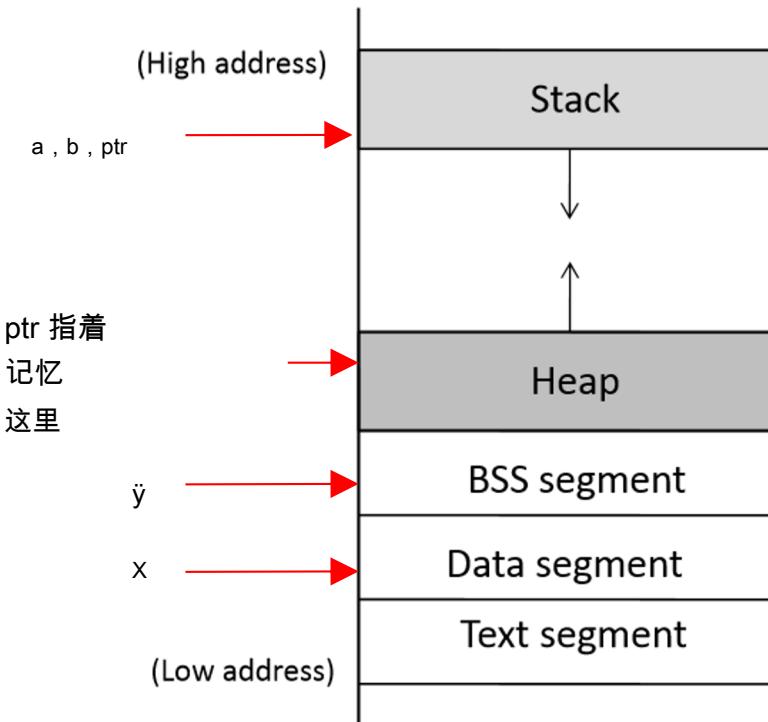
```
int x = 100;
int main()
{
    // data stored on stack
    int a=2;
    float b=2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2*sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    // deallocate memory on heap
    free(ptr);

    return 1;
}
```



函数参数的顺序

在堆栈中

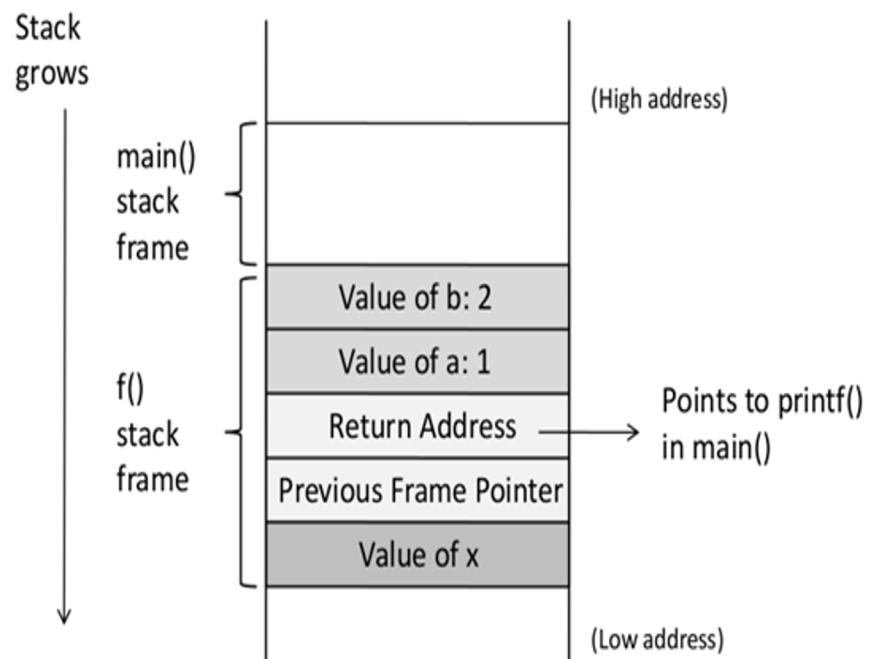
```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```

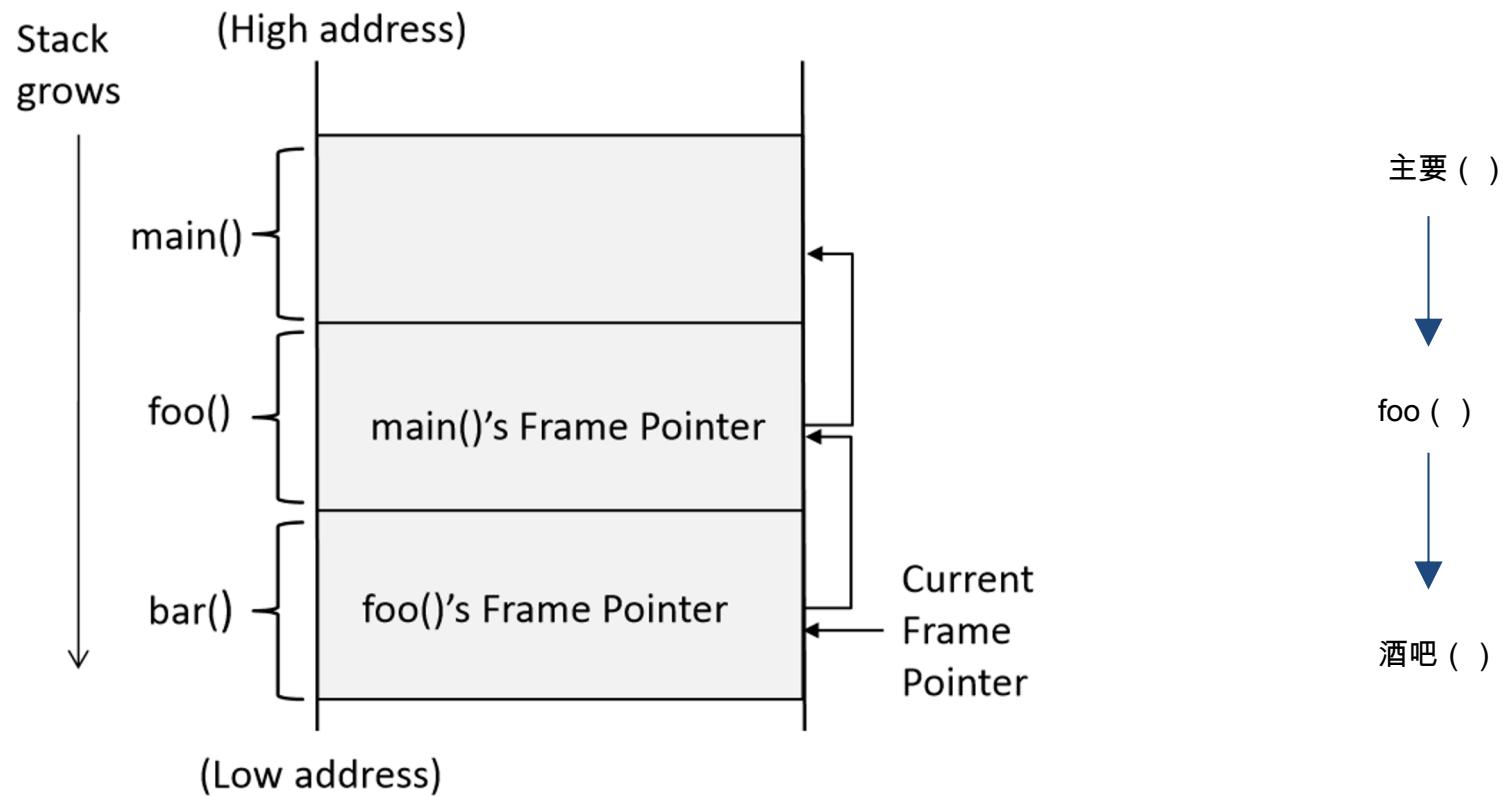
```
movl  12(%ebp), %eax      ; b is stored in %ebp + 12
movl  8(%ebp), %edx       ; a is stored in %ebp + 8
addl  %edx, %eax
movl  %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

函数调用栈

```
无效f ( int a , int b ) {  
  
    int x;  
}  
  
无效main ( )  
{  
    f ( 1,2 ) ;  
    printf ( " hello world" ) ;  
}
```



函数调用链的堆栈布局



堆栈溢出

- 堆栈溢出可能是由于：
 - 递归调用
 - 重入中断
 - 堆栈上的分配
 - 大
 - 由攻击者控制
 - Alloca(), char数组[function_parameter]

堆栈溢出

- 这永远不会发生？
- 检测：防护页（ Unix ）
 - 分配太大时出现问题
 - 堆栈指针可以“跳过”保护页面
- 预防：不太容易
 - 摘要解释[*Regehr05*]
 - 只要可以静态确定控制流即可工作
 - 禁止/绑定递归
 - 禁止/限制分配堆栈
- 微控制器问题
 - 没有MMU（ 内存管理单元 ）

堆栈缓冲区溢出/溢出

- 缓冲区溢出的特殊情况
 - 利用现有漏洞来修改控制流
- 溢出的缓冲区分配在堆栈上，通常会破坏返回地址
- 第一个大规模利用：Unix上的Morris蠕虫（1989）

弱势程序

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str); ←

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

- 从badfile读取300字节的数据。
- 将文件内容存储到大小为400字节的str变量中。
- 以str作为参数调用foo函数。

注意：Badfile由用户创建，因此内容由用户控制。
。

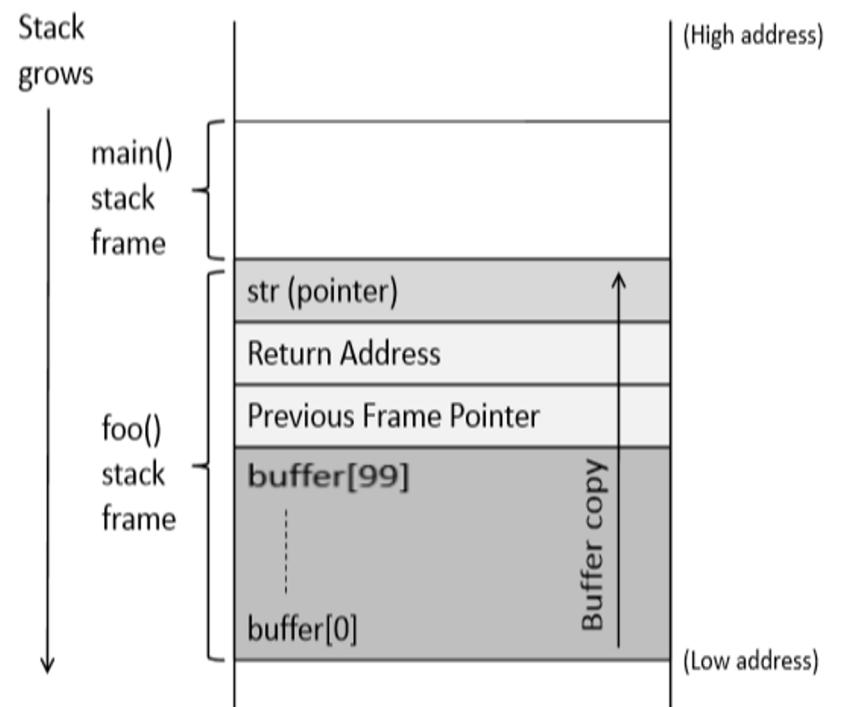
弱势程序

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

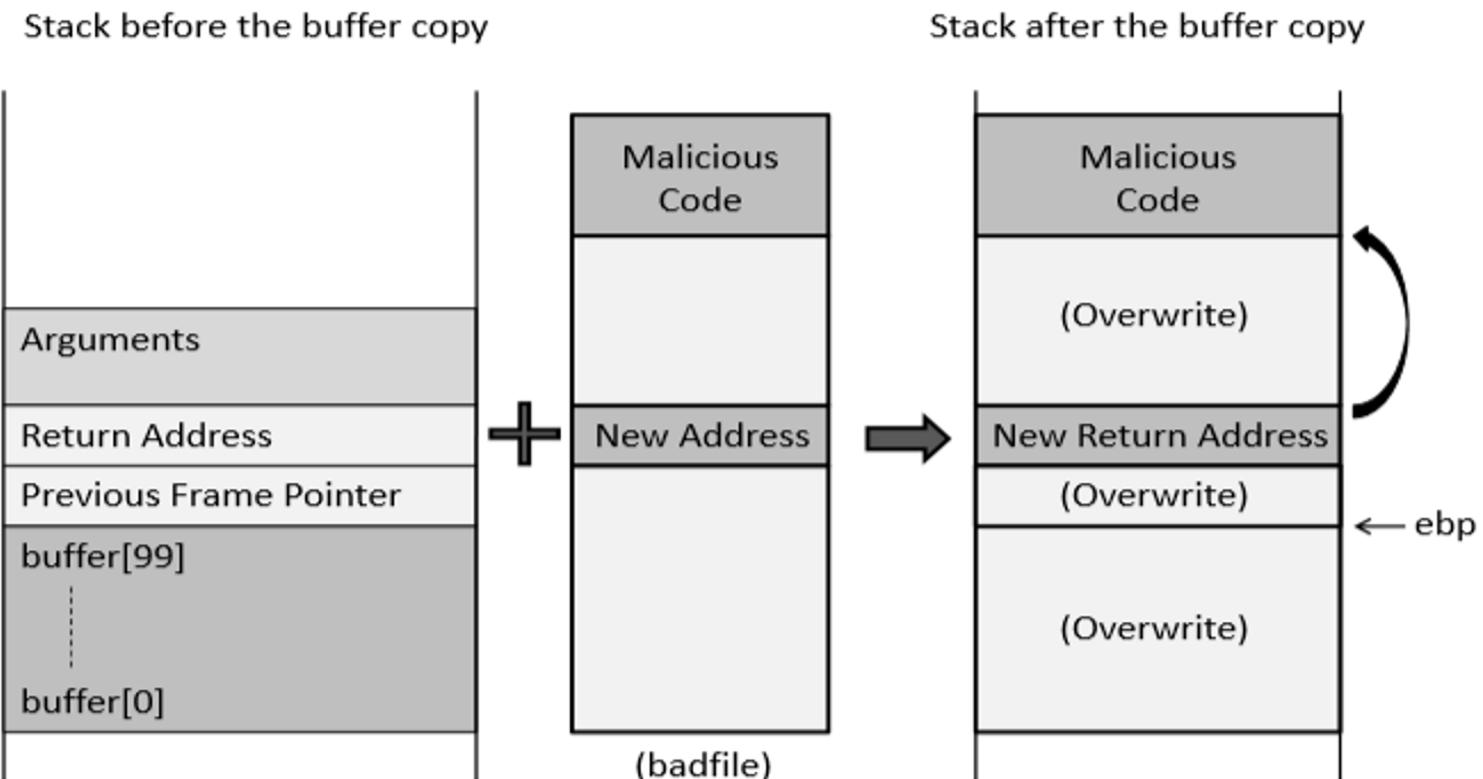


缓冲区溢出的后果

用一些随机地址覆盖返回地址可以指向：

- 无效的指令
 - 不存在的地址
 - 访问冲突
 - 攻击者代码
- 恶意代码获得访问权限

如何运行恶意代码

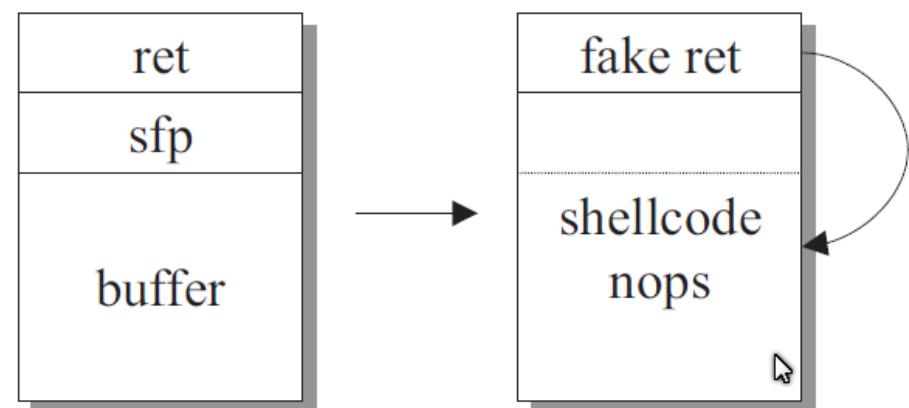
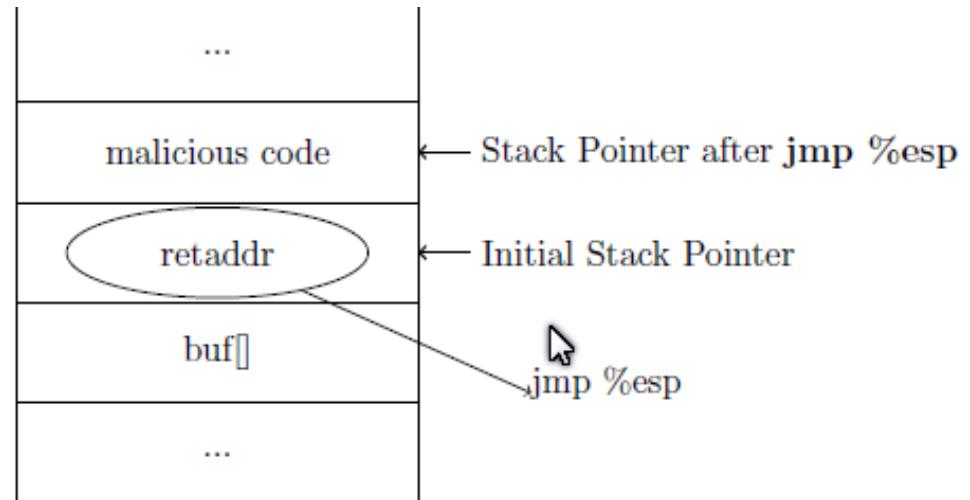


有关修改寄信人地址的更多信息

- 并非总是容易知道确切的地址

- 蹦床：
 - `jmp %esp` 固定地址

- NOP雪橇
 - 长时间的NOP
 - 后跟shellcode
 - 跳到那里的任何地方都会导致shellcode

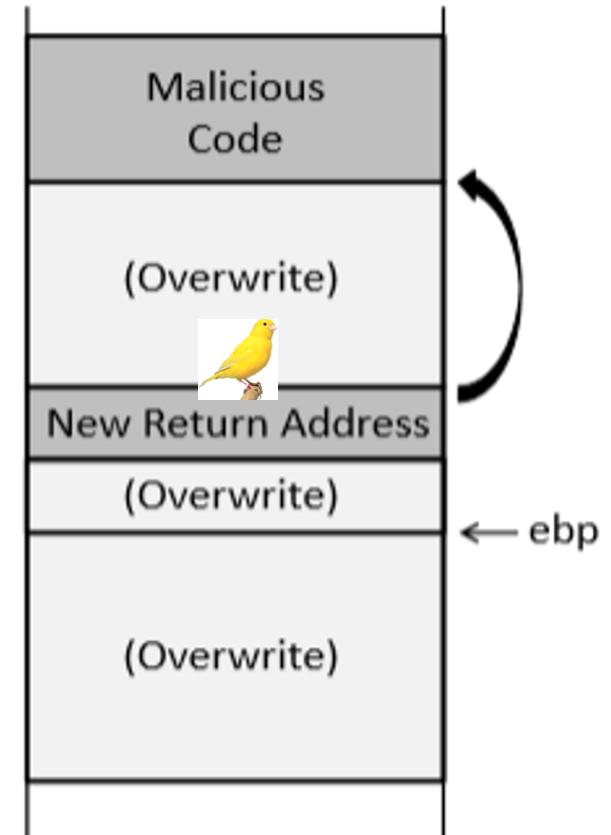


利用堆栈缓冲区溢出

- 破坏控制流程
 - 堆栈框架中的返回地址
 - 功能指针
 - 异常处理程序
 - 全局偏移表 (GOT)
 - C ++对象
- 损坏程序状态 (其他堆栈框架)
 - 存储身份验证状态的变量
 - 修改存储的密码
 - 指针值

保护：金丝雀

- 目标：检测意外修改堆栈上的值（例如，返回地址）
- 在堆栈上插入一个已知值
 - “金丝雀”
- 编译器扩展-插入以下代码：
 - 在返回地址后添加一个随机值
 - 在使用寄信人地址之前检查金丝雀值
- 局限性
 - 金丝雀可以通过内存泄漏来猜测或获得
 - 金丝雀副本需要存储在可能也已损坏的安全位置



保护：不可执行的内存

(可写XOR可执行文件/ NX)

- 可执行和可写存储区域
 - 允许直接编写指令（在缓冲区中）并执行它们，例如，从堆栈中。
 - 使基于堆栈的缓冲区溢出易于利用
- W XOR X (又名NX / XN / DEP / PAX)
 - 操作系统拆分内存区域—没有区域既可写又可执行
- 现在，大多数系统都支持W XOR X的形式：
 - 在x86上比较复杂（分段）
 - 自2004年以来HW对其的支持
- 相同功能的许多名称：
 - AMD：增强的病毒防护，
 - ARM：XN eXecute Never
- 受到“返回libc攻击” /“面向返回的编程”的打击

保护 : ASLR (地址 空间布局随机化)

- 随机化内存段的基地址
 - 地址在每次执行时都会更改
- 随机化以下项的起始或基地址：
 - 程式码
 - 库代码
 - 堆/堆栈/数据区域
- 目标：
 - 难以猜测内存中的堆栈地址。
 - 难以猜测 %ebp 地址和恶意代码的地址
- 许多程序都有问题
 - 有时需要重写其中的一部分，特别是当它们包含特制的汇编代码时
 - 代码需要与程序的位置无关的位置
- 局限性
 - 内存泄漏用于学习内存布局
 - 地址空间/系统限制（例如，页面边界）可能允许蛮力探测

地址空间布局 随机化

```
#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);
}
```

地址空间布局 随机化：工作

```
$ 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

```
$ 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
```

3