

实验原理

本次实验要攻击的是一个可执行程序 `bufbomb`。在开始动手前，我们需要先理解一些基础知识，就像拆弹专家需要先看懂炸弹结构一样。让我们先从最基础的概念开始吧！

本次实验围绕《深入理解计算机系统》一书的第三章内容展开，建议各位同学在进行实验前重温该章节内容，以确保对相关知识有深入的理解和掌握。为便于同学们迅速回顾关键知识点，下面将针对实验涉及的内容进行简要的梳理与提炼。

1. 可执行程序与反汇编

1.1 什么是可执行程序？

简单来说，就是将代码（像 `bufbomb.c`、`stack.c` 这类 `.c` 或 `.h` 源代码）编译链接后能让计算机直接运行的文件（`bufbomb`），运行时操作系统会把它加载到内存，按指令顺序干活。而生成的 `bufbomb` 文件，里面全是 0 和 1 组成的二进制代码。计算机能轻松读懂并执行这些代码，但我们人类看这些就像看天书📖。

举个栗子🌰：

假设有一个指令的二进制是 `0x8d 45 e0`，计算机看到后会说：“啊，这是让我把某个地址的数据装进寄存器！”而我们人类只会一脸懵：“这啥玩意儿？”

1.2 反汇编

为了让人类也能理解，我们需要一个翻译工具——反汇编器，它能把二进制代码转换成汇编指令。

举个实战操作：生成反汇编代码（用 `objdump` 工具）：

```
objdump -d bufbomb > bufbomb.s
```


这个命令会把 `bufbomb` 程序的机器码翻译成汇编代码，保存到 `bufbomb.s` 文件中。

查看关键函数：

```
00000000004013c3 <smoke>:  
4013c3: 55                push    %rbp
```

```
4013c4:  48 89 e5                mov     %rsp,%rbp
4013c7:  bf 08 30 40 00         mov     $0x403008,%edi
4013cc:  e8 9f fc ff ff         call    401070 <puts@plt>
4013d1:  bf 00 00 00 00         mov     $0x0,%edi
4013d6:  e8 0f 0a 00 00         call    401dea <validate>
4013db:  bf 00 00 00 00         mov     $0x0,%edi
4013e0:  e8 1b fe ff ff         call    401200 <exit@plt>
```

2. 基础汇编语言知识


 **为什么需要学汇编？**

汇编语言就像计算机的“方言”，虽然有点难懂，但学会后你就是计算机的贴身翻译官！

- **没有源代码：**就像拆弹时没有设计图纸，只能通过观察炸弹外壳（汇编代码）推测内部结构。
- **精准控制：**汇编直接对应 CPU 的每一步操作，比如：
 - `mov %rax, %rbx` 表示把 `rax` 寄存器的值复制到 `rbx`
 - `jne 401200` 表示如果就不跳转到 `0x401200` 地址

2.1 x86_64 寄存器结构

寄存器是CPU内部的小型存储单元，就像你手边的工具箱，存放着当前正在使用的工具（数据）。

 小知识：寄存器还有“缩小版”！比如 `%eax` 是 `%rax` 的低32位，`%al` 是低8位。

64位寄存器	低32位	用途说明
<code>%rax</code>	<code>%eax</code>	返回值、算术运算结果
<code>%rbx</code>	<code>%ebx</code>	基址寄存器、保存临时数据
<code>%rcx</code>	<code>%ecx</code>	计数器（如循环次数），函数第4参数
<code>%rdx</code>	<code>%edx</code>	数据寄存器、辅助算术运算，函数第3参数
<code>%rsi</code>	<code>%esi</code>	源索引（如字符串操作源地址），函数第2参数

%rdi	%edi	目的索引（如字符串操作目的地址），函数第1参数
%rbp	%ebp	基址指针（栈帧基地址）
%rsp	%esp	栈指针（当前栈顶地址）
%r8	%r8d	通用寄存器，函数第5参数
%r9	%r9d	通用寄存器，函数第6参数
%r10-%r15	%r10d-%r15d	通用寄存器，临时数据存储

下图来自CSAPP P120：

63	31	15	7	0	
%rax	%eax	%ax	%al		返回值
%rbx	%ebx	%bx	%bl		被调用者保存
%rcx	%ecx	%cx	%cl		第4个参数
%rdx	%edx	%dx	%dl		第3个参数
%rsi	%esi	%si	%sil		第2个参数
%rdi	%edi	%di	%dil		第1个参数
%rbp	%ebp	%bp	%bpl		被调用者保存
%rsp	%esp	%sp	%spl		栈指针
%r8	%r8d	%r8w	%r8b		第5个参数
%r9	%r9d	%r9w	%r9b		第6个参数
%r10	%r10d	%r10w	%r10b		调用者保存
%r11	%r11d	%r11w	%r11b		调用者保存
%r12	%r12d	%r12w	%r12b		被调用者保存
%r13	%r13d	%r13w	%r13b		被调用者保存
%r14	%r14d	%r14w	%r14b		被调用者保存
%r15	%r15d	%r15w	%r15b		被调用者保存

2.2 常用汇编指令（ATT格式）

这些指令是你攻击缓冲区时的瑞士军刀，一定要认准 **ATT格式** 的“刀柄”（语法）！



什么是ATT（AT&T）格式

ATT（AT&T）格式是一种汇编语言语法规则，常用于 Unix 和类 Unix 系统（如 Linux）中，是 GCC 编译器和 objdump 工具默认使用的汇编格式。除了 ATT 格式，常见的汇编语言语法规则还有 Intel 格式、MASM 格式、NASM 格式等。在 Windows 系统和一些 x86 架构的开发环境中，许多商业软件开发工具（如 Visual Studio 中的汇编器）默认采用 Intel 格式。

2.2.1 高频汇编指令表

指令	功能说明	示例	实战场景
mov	数据搬运	mov \$0x10, %rax	把数字0x10装进%rax
add	加法	add %rbx, %rax	计算 %rax = %rax + %rbx
cmp	比较	cmp \$0x5, %rax	比较 %rax - 5 的值，影响后续跳转
jmp	无条件跳转	jmp 0x401200	强制跳转到0x401200地址执行
call	打电话呼叫函数	call	调用字符串比较函数
ret	挂电话返回	ret	结束当前函数，回到调用位置
lea	地址计算	lea -0x20(%rbp), %rax	计算 %rbp-0x20 的地址存入%rax
movzbl	零扩展搬运	movzbl (%rdi), %eax	读取输入字符串的第一个字符（转成整型）ASCII值
and	比特位	and \$0xf, %eax	取字符低4位作为数组索引 (phase_5)
push/pop	栈操作	push %rbx	保存寄存器原始值到栈里

2.2.2 跳转指令表

指令	英文全称	中文含义	检查的标志位组合	典型应用场景
jne	Jump if Not	不等时跳转	ZF=0	密码错误时引

Equal				爆炸弹💣
je	Jump if Equal	相等时跳转	ZF=1	输入匹配预设值后继续执行 ✅
jg	Jump if Greater	有符号数大于	SF=OF 且 ZF=0	检查输入数值是否超过阈值 📈
jge	Jump if Greater or Equal	有符号数大于等于	SF=OF 或 ZF=1	验证数组索引是否合法✅
jl	Jump if Less	有符号数小于	SF≠OF 且 ZF=0	检查密码长度是否不足
jle	Jump if Less or Equal	有符号数小于等于	SF≠OF 或 ZF=1	循环终止条件 ($i \leq N$)
ja	Jump if Above	无符号数大于	CF=0 且 ZF=0	内存地址越界检查🚫
jae	Jump if Above or Equal	无符号数大于等于	CF=0 或 ZF=1	缓冲区大小验证✅
jb	Jump if Below	无符号数小于	CF=1 且 ZF=0	检查数组索引是否越界
jbe	Jump if Below or Equal	无符号数小于等于	CF=1 或 ZF=1	输入字符范围限制（如0-9）

2.3 ATT 格式的特征

ATT 格式的语法有以下几个标志性特征，让它在众多汇编“方言”中独树一帜：

- 操作数顺序：源→目的（和直觉相反！和 Intel 格式完全相反！）

语法规则：指令 源操作数，目的操作数

```
mov $0x10, %rax    # 把数字 0x10 (源) 复制到 %rax (目的)
add %rbx, %rax     # 计算 %rax = %rax (目的) + %rbx (源)
```

- 寄存器前缀: % 符号

语法规则: 所有寄存器名前加 %

```
mov %rax, %rbx    # 正确写法 (ATT格式)
mov rax, rbx      # 错误! (Intel格式, 会报错)
```

- 立即数前缀: \$ 符号

语法规则: 立即数 (常数) 前加 \$

```
mov $0x10, %rax    # 把数字 0x10 存入 %rax
cmp $5, %eax       # 比较 %eax - 5 的值
```

- 内存寻址: 括号 (地址计算器)

语法规则: 偏移量 (基址寄存器, 索引寄存器, 比例因子)

```
mov (%rbx), %rax    # 从地址 %rbx 处读取值 → 相当于 *rbx
mov 0x8(%rbp,%rcx,4), %eax # 计算地址 = %rbp + %rcx*4 + 8, 读取该地址的值
```

- 指令后缀: 数据大小标记 (防呆设计)

语法规则: 指令后加后缀标明操作数大小:

```
b: 字节 (8位) → movb
w: 字 (16位) → movw
l: 双字 (32位) → movl
q: 四字 (64位) → movq
```

```
movb $0x1, %al     # 移动 1 字节到 %al
movq %rax, %rbx    # 移动 8 字节 (64位)
```

后缀像刀柄上的刻度, 告诉 CPU: “这次操作切多宽的数据块!”

实战急救包: ATT vs Intel 对照表

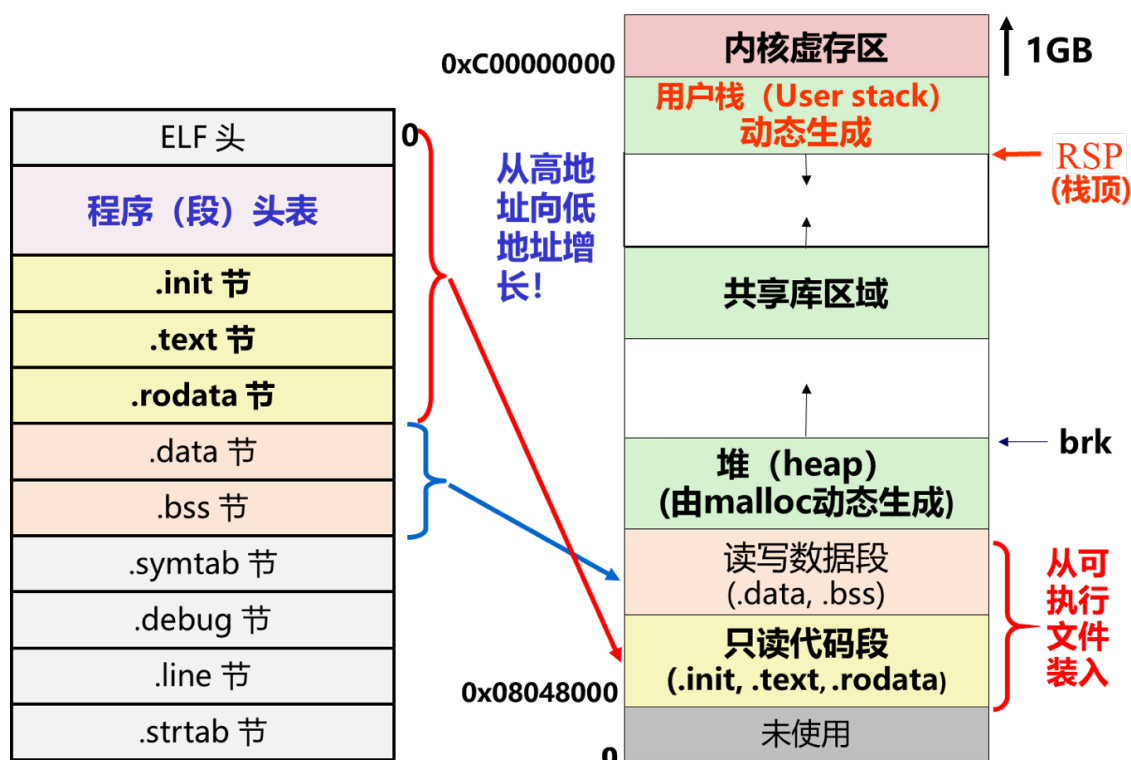
场景	ATT 格式	Intel 格式

移动数据	mov \$0x10, %rax	mov rax, 10h
比较数值	cmp %rax, %rbx	cmp rbx, rax
读取内存	mov 0x8(%rbp), %eax	mov eax, [rbp+8]
函数调用	call 0x401000	call 0x401000（格式相同）

3 过程调用的机器级表示

3.1 可执行文件的存储器映像

为了统一，模块代码之间必须遵循调用接口约定，称为调用约定(calling convention)，具体由ABI规范定义，编译器强制执行，汇编语言程序员也必须强制按照这些约定执行，包括寄存器的使用、栈帧的建立和参数传递等。



3.2 过程（函数）的结构

一个C过程的大致结构如下：

- (1) 准备阶段

- 1) 形成帧底: push指令 和 mov指令
- 2) 生成栈帧 (如果需要的话): sub指令 或 and指令
- 3) 保存现场 (如果有被调用者保存寄存器): push指令
- (2) 过程 (函数) 体
 - 1) 分配局部变量空间, 并赋值
 - 2) 具体处理逻辑, 如果遇到函数调用时
 - 准备参数: 将实参送栈帧入口参数处, 例如:

```
movq  参数3, 8(%rsp)
.....
movq  参数1, (%rsp)
```

- CALL指令: 保存返回地址 (即call指令的下一条指令的地址) 并转被调用函数, 例如:

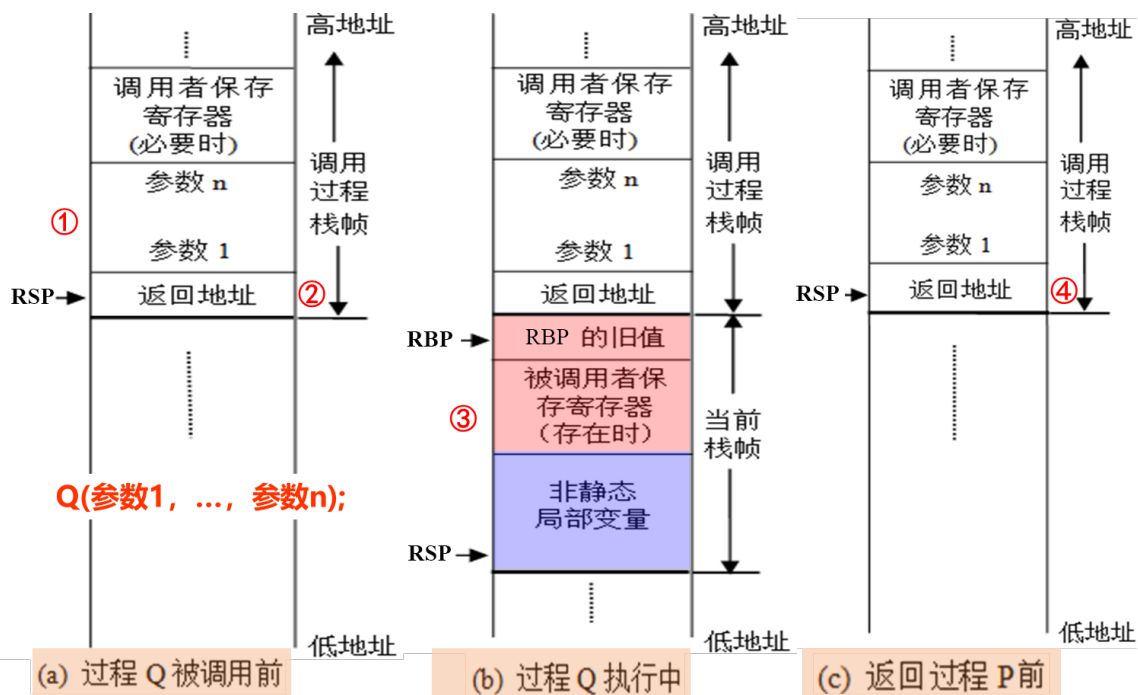
```
call  add
```

相当于

```
R[rsp] ← R[rsp] - 8
M[R[rsp]] ← 返回地址
R[rip] ← add函数首地址
```

- 3) 在RAX中准备返回参数
- (3) 结束阶段
 - 1) 退栈: leave指令 或 pop指令。leave 指令实际上是 movq %rbp, %rsp 和 popq %rbp 两条指令的组合, 用于快速恢复栈帧。
 - 2) 取返回地址返回: ret指令

3.2 过程调用过程中栈和栈帧的变化



过程调用的执行步骤如下（P为调用者，Q为被调用者）：

- (1) P将入口参数（实参）放到Q能访问到的地方；
- (2) P保存返回地址，然后将控制转移到Q（CALL指令）；
- (3) Q保存P的现场，并为自己的非静态局部变量分配空间；
- (4) 执行Q的过程体（函数体）；
- (5) Q恢复P的现场，释放局部变量空间；
- (6) Q取出返回地址，将控制转移到P（RET指令）。

其中（1）和（2）是P过程，（3）~（6）是Q过程。