目录

[第2讲 软件安全（1）系统模型：从源代码到执行 2](#_Toc12810787)

[第2讲 软件安全（2）控制流劫持攻击 4](#_Toc12810788)

[第2讲 软件安全（4）竞态条件 8](#_Toc12810789)

[第3讲 信息安全模型 12](#_Toc12810790)

[第4讲 安全体系结构（2） 14](#_Toc12810791)

[第6讲 Web安全 15](#_Toc12810792)

第2讲 软件安全（1）系统模型：从源代码到执行

**1 理解函数调用中堆栈工作原理。**

1. 每个函数都是一个栈结构，有一个栈底指针ebp和栈顶指针esp。栈底指针在函数的执行过程中是不变的，栈顶指针会随着函数的执行动态的增大。
2. 在操作系统中，栈底在地址最大的地方，栈是向地址小的方向增长的。所以在栈中，申请空间时，是将栈顶指针esp向下移动，即减去一个大小。
3. 函数的层层调用，就形成了连续的栈空间。
4. 函数在调用被调用函数时，会将需要传递的参数由从右向左的顺序入栈。被调用函数会根据这个关系去对应的位置取参数。
5. 将参数入栈之后，还会将下一条指令的地址入栈。即从被调用函数返回之后，应该执行哪一条指令。
6. 跳转至被调用函数。被调用函数需要将当前的栈底指针ebp的值入栈，因为当被调用函数执行完毕之后，需要恢复调用函数的栈空间。并将ebp寄存器的值设为当前函数的栈空间的底部对应的地址，即当前esp的值。
7. 被调用函数执行时，根据参数的传入顺序便可以计算出参数相对于ebp指针的位置，从而获取变量的值。
8. 被调用函数执行完毕后，如果有返回值，会将返回值存入eax寄存器中。并释放申请的栈空间，恢复ebp和esp的值。从而使被调用函数能够正常执行。
9. 将指针寄存器指向调用函数压入栈中的返回地址。Ret：用栈中的数据，修改IP的内容，实现近转移，pop IP然后执行指令。Ret n： pop IP+add esp, n，然后执行指令。
10. 调用函数继续执行，如果需要结果，则从eax寄存器中获取

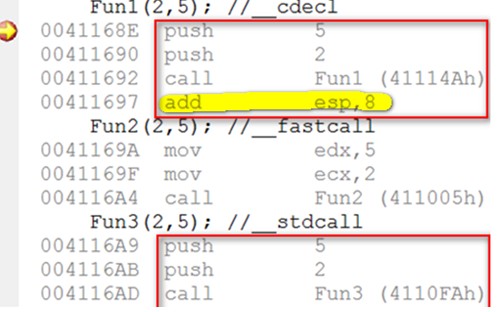
**2. 为什么需要调用约定?比较几种典型的调用约定?**

\_cdecl、\_\_stdcall、\_\_fastcall是C/C++里中经常见到的三种函数调用方式。其中\_\_cdecl是C/C++默认的调用方式，\_\_stdcall是windows API函数的调用方式，只不过我们在头文件里查看这些API的声明的时候是用了WINAPI的宏进行代替了，而这个宏其实就是\_\_stdcall了。

\_\_cdecl： C/C++默认方式，参数从右向左入栈，主调函数负责栈平衡。

\_\_stdcall：windows API默认方式，参数从右向左入栈，被调函数负责栈平衡。

\_\_fastcall：快速调用方式。所谓快速，这种方式选择将参数优先从寄存器传入（ECX和EDX），剩下的参数再从右向左从栈传入。因为栈是位于内存的区域，而寄存器位于CPU内，故存取方式快于内存，故其名曰“\_\_fastcall”。



\_cdecl按照参数从右向左的方式进入栈区，注意Fun1()和Fun3()的区别，Fun1()在call Fun1()之后执行了add esp,8。这一操作正是我们前面所说的进行栈的平衡。调用函数之前连续进行了两次push操作将函数所需的实参5和2先后压入了栈区，调用完成后，我们需要恢复调用前的状态，则需调整栈顶指针esp的位置，这一工作由谁来完成就决定了两种函数调用方式\_\_cdecl（主调函数完成）和\_\_stdcall（被调函数完成）的产生。上图我们看到了\_\_cdecl中由主调函数完成了。

\_\_stdcall，在被调函数Fun3()中，转向被调函数结尾处的代码：ret 8

——cdecl，在被调函数Fun1()结尾处，ret

——cdecl由函数调用者来清理，stdcall函数结束时自行清理

**3．了解可变参数varargs 如何工作?** **void printf(const char \*fmt, ...){}**

原理非常简单，就是从栈中取出数据而已，为了实现这一目的，必须通过第1个参数指定后续参数的数目和类型，这样我们才能解析出栈中的数据。这也就是为什么参数可变的函数，都至少带着一个有名形参，例如printf(const char\* fmt, ...)，这个有名形参由两个作用：

（1）用来指出后续形参的数目和类型，例如printf函数就是通过%X的形式来指定的，有多少个%就有多少个后续参数，参数的类型由%后面的标识符来指定，例如%c,%d,%X。实际上fmt字符串就是一套标准化的传输协议而已，你完全可以自定义一套你自己的实参解析协议。

（2）用来提供本函数的栈的首地址，只有借助这个首地址，才能根据偏移取出后续参数。换句话说，如果你拿不到本函数运行时栈的地址，根本就无法取参。

第2讲 软件安全（2）控制流劫持攻击

**1. 阅读strncpy(), strncat()等函数的代码，理解它们为什么是不安全的?**

strncpy()和strncat()函数与strcpy()和strcat()函数是类似的，但每个函数都有一个额外的size\_t类型的参数n用于限制要被复制的字符数量。这些函数可以被认为是截断型的复制和拼接函数。

strncpy()库函数与strcpy()完成相似的功能，但前者允许指定一个最大大小n：

**char \* strncpy(char \*dest, char \*src, size\_t n);**

将字符串src中最多n个字符复制到字符数组dest中(它并不像strcpy一样遇到NULL就开始复制，而是等凑够n个字符才开始复制），返回指向dest的指针。

**如果n>dest串长度**，dest栈空间溢出产生崩溃异常。

**如果src长度<dest长度（包含串尾NULL字符）**

如果n=(0,src串长度)，src的前n个字符复制到dest中。但是由于没有NULL字符，所以直接访问dest串会发生栈溢出的异常情况

如果n = src串长度，与strcpy一致。

如果n = dest串长度，[0,src串长度]处存放src字串，(src串长度, dest串长度]处存放NULL。

**如果src长度>dest长度（包含串尾NULL字符）**

如果n =dest串长度，则dest串没有NULL字符，会导致输出会有乱码。如果不考虑src串复制完整性，可以将dest最后一字符置为NULL。

**所以strncpy()函数不能保证用空字符终止目标字符串，所以程序员必须小心，以确保目标字符串是正确地以空字符终止的，并且没有覆盖最后一个字符。**

**strncat()函数**从s2指向的数组追加不超过n个字符（空字符和它后面的字符不追加）到s1指向的字符串结尾。s2最初的字符覆盖了s1末尾的空字符。终止空字符总是被附加到结果字符串。因此，在s1指向的数组中的最大字符数量是strlen(s1) + n + 1。

**限制长度，却又不保证'\0'结束**

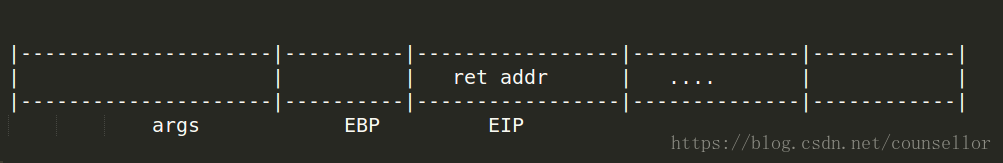
**2. 理解Return to libc的实现原理，理解ret2libc利用中栈的变化情况。**

return-to-libc 是一种对抗linux系统栈保护的攻击方法。我们知道大部分linux系统有栈保护机制(DEP)。简单的说，既然栈中的指令不能执行，我们可以找到libc中的函数去执行，这样就绕过了数据不可执行的问题了。

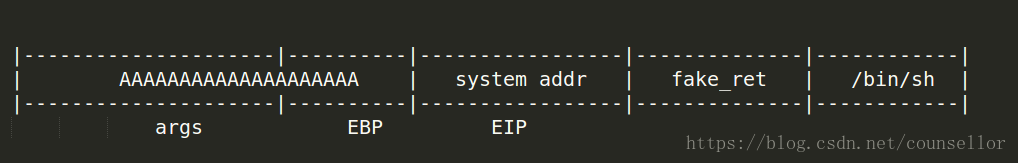
**攻击原理**

通过把函数返回地址直接指向系统库中的函数（如system函数），同时构造该函数的输入参数栈，就可以达到代码执行的目的。

正常堆栈布局：



ret2libc执行system的堆栈布局：



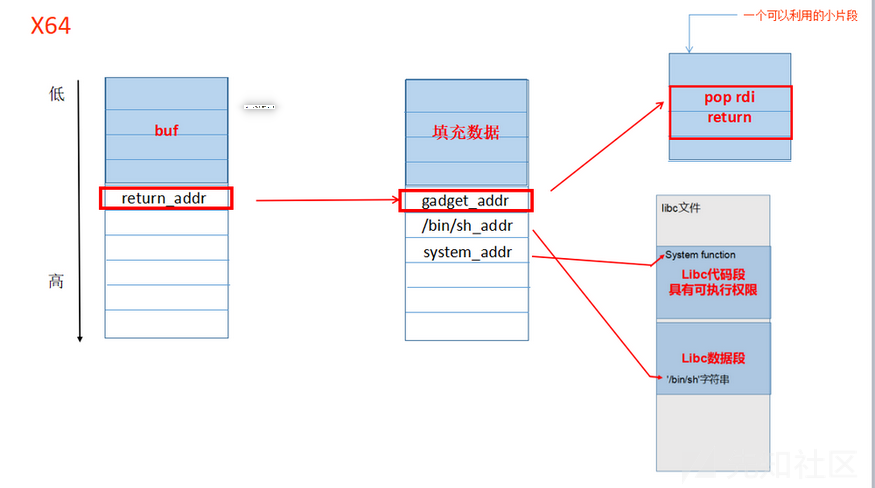
这样我们就获得了攻击payload的布局结构：

A\*N + system\_addr + fake\_ret + system\_arg

**3. 理解ROP的实现原理。**

ROP 绕过 NX 原理

这里引用别人的图片和说明。最基本的 ROP 攻击缓冲区溢出漏洞的原理：（图里基于 x64 平台，注意 x64 使用 rdi 寄存器传递第一个函数参数）



工作原理描述如下：

①当程序运行到 gadget\_addr 时（rsp 指向 gadget\_addr），接下来会跳转到小片段里执行命令，同时 rsp+8(rsp 指向 bin\_sh\_addr)

②然后执行 pop rdi, 将 bin\_sh\_addr 弹入 rdi 寄存器中，同时 rsp + 8(rsp 指向 system\_addr)

③执行 return 指令，因为这时 rsp 是指向 system\_addr 的，这时就会调用 system 函数，而参数是通过 rdi 传递的，也就是会将 /bin/sh 传入，从而实现调用 system('/bin/sh')

所以纵观我们整个 ROP 利用链的环节，有三个很重要的问题需要解决：

怎么去搜索这样的 gadget\_addr，当然不止一次 pop，还可以多个 pop 加 ret 组合等等，看你希望怎么去利用

如何得到 '/bin/sh\0' 这样的字符串，通常程序没有这样的字符串

如何得到 libc 中 system 实际运行的地址（libc 的基地址+system 在 libc 中的偏移地址）

其实还有一个问题很重要，就是确定你的返回地址 return\_addr 前面缓冲区到底有多大，这样才能准确的实现缓冲区溢出覆盖。做法有二种：一是直接从 IDA 的 F5 源码和汇编计算得到；二是使用 GDB 动态调试一下。

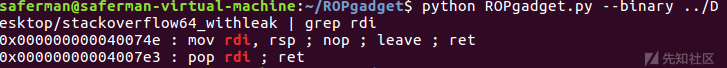
ROP 绕过 NX 具体步骤

下面分别就上面三个问题给出具体的解决方案，最终完成整个 ROP 绕过 NX 保护的攻击。

（1）如何搜索你需要的 gadget\_addr？

gadget\_addr 指向的是程序中可以利用的小片段汇编代码，在上图的示例中使用的是 pop rdi ; ret ;

对于这种搜索，我们可以使用一个工具：ROPgadget



图中地址 0x4007e3 就是我们需要的 gadget\_addr。

（2）bin\_sh\_addr 指向的是字符串参数：/bin/sh\0。

首先你需要搜索一下程序是否有这样的字符串，但是通常情况下是没有的。这时候就需要我们在程序某处写入这样的字符串供我们利用。我们需要用 IDA 打开程序，看左边函数窗口程序加载了下面哪些函数：

read、scanf、gets

通常我们将这个字符写入 .bss 段。.bss 段是用来保存全局变量的值的，地址固定，并且可以读可写。通过 readelf -S pwnme 这个命令就可以获取到 bss 段的地址了（ida 的 segements 也可以查看）。

（3）system\_addr 则指向 libc 中的 system 函数 。

可以先查看一下程序本身有木有可以利用的子函数。因为从 libc 中使用函数，需要知道 libc 的基地址。通常得到 libc 基地址思路就是：

泄露一个 libc 函数的真实地址 => 由于给了 libc.so 文件知道相对偏移地址 => libc 基地址 => 其他任何 libc 函数真实地址

泄露一个 libc 函数的地址需要使用一个能输出的函数，同样用 IDA 打开程序，看左边的函数窗口程序加载了哪些函数可以利用：

write、printf、puts

特别注意：由于 libc 的延迟绑定机制，我们需要选择已经执行过的函数来进行泄露。你需要找到函数的 plt 地址，找到 jmp 指向的那个地址才是我们需要泄露的。

用脚本查找bin/sh和system

from pwn import \*

p = process('./pwnme')

p.recvuntil('shellcode')

elf = ELF('libc.so.6')

system\_in\_libc = elf.symbols['system']

bin\_sh\_in\_libc = next(elf.search('/bin/sh'))

libc\_base = 0x00007ffff7dee000

gadget\_addr = 0x0000000000401243

system\_addr = libc\_base + system\_in\_libc

bin\_sh\_addr = libc\_base + bin\_sh\_in\_libc

print hex(system\_addr) +'----'+hex(bin\_sh\_addr)

**4. 理解控制流劫持的攻击方式和防御方式**

**三种攻击实例**

缓冲区溢出攻击

整数溢出攻击

格式化字符串漏洞

**攻击方式**：Shell code注入、ret2libc、ROP

**防御方式:** NX、StackGuard、ASLR

NX：允许溢出，防止shellcode执行

添加运行时检查代码，检测溢出利用，当检测到进程被攻击者用溢出漏洞攻击时挂起该进程

StackGuard：运行时检测栈的完整性，在栈帧中嵌入“canaries”，在函数返回前校验“canaries”是否被改变，从而保证栈的完整性

StackShield：函数调用前，将返回地址RET和ebp复制到“安全”位置，函数返回前，恢复保存的RET和ebp

ASLR：将共享库映射到进程内存中的随机位置，攻击者不能直接跳转到exec函数

其他随机化方法:

系统调用随机化: 随机化系统调用的ID

指令集合随机化 (ISR)

第2讲 软件安全（4）竞态条件

1. **说明dirtyCOW的攻击原理，并说明dirtyCOW攻击有哪些后果？**

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。

**基本思路**: 需要运行两个线程

线程1: 使用write()写入映射的内存

线程2: 丢弃映射内存的私有副本

需要使得这两个进程相互竞争，以便它们能影响输出

**漏洞影响**

可以非法获得读写权限，进而提升权限；这种状态下也就等于电脑虽然在你这，但已经不归你控制了

攻击非常便利，而且成功率很高

杀毒软件无效

1. **调研并理解其它 side-channel attacks 方法；**

·边信道攻击(side channel attack 简称SCA)，又称侧信道攻击:针对加密[电子设备](https://baike.baidu.com/item/%E7%94%B5%E5%AD%90%E8%AE%BE%E5%A4%87/4393826)在运行过程中的时间消耗、功率消耗或电磁辐射之类的侧信道信息泄露而对加密设备进行攻击的方法被称为边信道攻击。这类新型攻击的有效性远高于密码分析的数学方法，因此给密码设备带来了严重的威胁。

**方法**

边信道攻击方法 主要集中在功耗攻击、电磁场攻击和时间攻击。其中功耗攻击是最强有力的手段之一，包括简单功耗分析攻击（SimplePower Analysis attacks,SPA）和差分功耗分析攻击（Differential　Power　Analysis　attacks,DPA），与传统密码分析学相比，这些攻击手段攻击效果显著。

**实现**

几种已实现的边信道攻击

1.通过CPU缓存来监视用户在浏览器中进行的快捷键及鼠标操作

对最新型号的英特尔CPU有效，如Core i7；还需运行在支持HTML5的浏览器上。带有恶意JS的网页在受害者电脑上执行后，会收集与之并行的其它进程的信息，有了这个信息，攻击者可以绘制内存对按下按键和鼠标移动的反应情况，进而重塑用户使用情景。

2.“听译”电子邮件密钥

通过智能手机从运行PGP程序的计算机中“听译”密钥。这项最新的密钥提取攻击技术，能够准确地捕捉计算机CPU解码加密信息时的高频声音，并提取密钥。

3.非智能手机+恶意软件+目标PC

从采购供应链下手，将特制小体量难以检测的恶意软件植入电脑，该软件会强制计算机的内存总线成为天线，通过蜂窝频率将数据无线传输到手机上。攻击者将接受和处理信号的软件嵌入在手机的固件基带中，这种软件可以通过社会工程攻击、恶意App或者直接物理接触目标电话来安装。

4.用手触碰电脑即可破解密码

电脑CPU运算时造成“地”电势的波动，用手触碰笔记本电脑的外壳，接着再测量释放到皮肤上的电势，然后用复杂的软件进行分析，最终得到计算机正在处理的数据。例如：当加密软件使用密钥解密时，监测这种波动就可得到密钥。

5.智能手机上的FM无线电功能来拾取电脑显卡发出的无线电波

6.利用KVM入侵物理隔离设备

使用连接到互联网的设备下载恶意软件，然后将其传递给设备的内存。之后透过KVM漏洞传播给使用KVM操控的其它多台设备，实现入侵物理隔离的系统，并感染更敏感的设备。最后恶意程序再经KVM反向将窃取到的数据传递到互联网。

7.利用一个面包（皮塔饼）偷取计算机密钥

无屏蔽铜线圈、电容

8.通过热量窃取电脑信息

9.其它方法

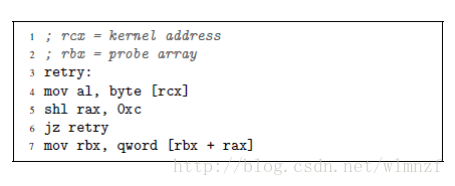
分析设备在解密过程中的内存利用率或放射的无线电信号，窃取密钥。

1. **说明Meltdown的攻击原理；**



  在第一行，我们引发了一个异常，若程序按照顺序执行，第三行的access将不会被执行，但是由于程序是乱序执行的，因此在异常引发之前，第三行的access将被执行，但是执行结果存在ROB中不会被提交，当异常触发以后，第三行的执行结果将被撤销掉，从计算机外部看来第三行的access就跟从未被执行过一样，但是事实上它只是在后期被撤销了，这会带来一个小问题，就是在执行过程中，刚才读取的数据已经被存储到高速缓存中。

我们可以通过侧信道攻击来确认刚才访问的数据是什么。



这段代码揭示了攻击的关键部分

首先我们将访问内核地址并取内容到寄存器内，

若这一举动可以触发异常，则对应的寄存器清零，若异常未处理，则应用程序将会被终止，并且取出的值将会存入应用程序核心转储的寄存器中。我们要做的就是通过第5行将相应的寄存器清零，从而可以判断这是一个错误的值。然后再次重试，直到遇到一个不为0的值，然后将秘密的值作为地址，去做读取操作，以便cache记录到这个地址，实现侧信道攻击。

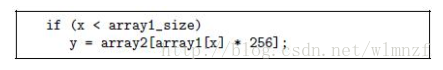
1. **说明防御Meltdown攻击的方法；**

V3的缓解

抵御Meltdown攻击最有效的方式就是KAISER/KPTI。KAISER/KPTI方案中要求操作系统维护两个页表，一个页表给用户程序使用，一个给kernel自己使用。并且确保程序所使用的页表不会映射高优先级的页面，即不会映射kernel的页面。KAISER/KPTI方案最早提出时是为了侧信道攻击对内核地址随机化（KASLR）的影响。该方案恰巧也可以用来抵御Meltdown攻击。

两个页表的切换，会导致CR3的重新加载，从而引起TLB刷新，进而降低内存的访问速度。如果某些应用场景需要大量的内核和用户空间切换（两个页表之间的切换），会造成较高的性能开销。为了降低这些性能开销，kernel需要使用充分的利用PCID特性，把TLB的刷新降低。

1. **说明Spectre的攻击原理；**



Spectre主要利用了分支预测和乱序执行的漏洞实现的，如图所示的代码，看起来十分地正常，若x小于array1的长度的时候，循环顺利执行。但是我们假设这里有一个存储密码的变量地址为secret，并且令a=secret-array1，于是我们可以使用array1[a]来表示secret的值。当多次执行循环的时候，我们的x满足循环条件，则我们的分支预测模块会认为下一个循环也满足循环条件而去预执行这个循环，若此时我们将a的值赋值给x，则分支预测模块预测本次循环为执行（其实并不会执行），CPU会预执行这个循环体，然后将我们存储密码的secret值取出来，并将其作为地址去访问array2，但是最终发现循环不应该被执行，于是刚才取出来的值将会被作废。但是我们的这个secret值的地址已经被存入到缓存中去。我们最终可以将array2读取一遍，若读取某个地址的时候，访问的时间特别短，则说明这个地址就是那个被存入缓存的地址（即我们的密码值）。

第3讲 信息安全模型

1. **分析BLP存在的隐蔽通道问题的原理。**

BLP模型的强制访问控制可以概括为不允许“上读，下写”，这种规则是由信息的保密性的安全要求决定的。保密性要求只有高保密级的主体能够读低保密级客体的内容，否则会造成高保密级的客体的信息泄密；反过来，高保密级的主体对低保密级的客体进行写操作也会造成信息泄密。

BLP的安全目标

高安全级别信息无法流向低安全级别的用户

通过公开通道（例如，读/写对象）的非法信息流被BLP阻止

但是，通过隐蔽通道的非法信息流仍然可能发生

基于系统资源使用的通信通道，而这些通道通常不用于系统中的主体（进程）之间的通信

1. **说明C-W模型的实施规则是从哪几个方面进行完整性访问控制的？它的证明模型是从哪几个方面确保完整性访问控制的有效性的？**

Clark-Wilson模型简称CWM。其核心目的是为了防止合法用户做非法操作，核心思想：

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

C-W模型采取两类措施来支持系统完整性

系统实施的措施 （实施规则，ER）

ER1 CDI仅由授权的TP更改

er2 tp仅由授权用户运行

ER3用户已通过身份验证

ER4授权仅由认证机构更改

用于证明系统实施的有效性的措施 （证明规则，CR）

CR1 IVPS验证CDI完整性

CR2 TPS保持CDI完整性

CR3 ER2职责分离

CR4 TPS写入日志

CR5 TPS将UDIS升级为CDIS

主要目标：防止利益冲突(conflicts of interest)

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：

1. **分析访问控制矩阵和域定义表的相同和不同之处。**

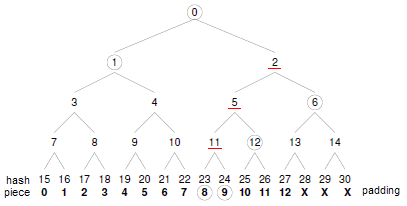
域定义表”(Domain Definition Table, DDT)，描述各个域对不同型客体的访问权限。

访问控制矩阵（Access Control Matrix）,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。

1. **莫科尔树模型的目标是以较少的内存开销实现较快的数据完整性验证，请以其某种应用为例，分析它是如何实现这一目标的。**

大家都知道，目前BT应用的发展具有一个非常显著的趋势，那就是用来交换电影、游戏、ISO等大尺寸的数据文件。然而我们也能够观察到另一个事实，那就是：下载文件所对应的索引文件(.torrent)也越来越大，越来越难以下载；这是因为在索引文件中保存了被下载文件中所有数据块的20字节SHA1校验值，而文件越大，数据块越多，则.torrent文件越长(块数=文件长度/数据块长，Bit Torrent标准协议建议块长一般不超过512KB)。索引文件长度的膨胀将直接加重索引服务器的下载负担，因为所有BT用户均必须从服务器上下载同一份.torrent文件，而这一下载行为本身是集中式的，因而导致系统的扩展性受到较大限制.

Merkle哈希树校验方式为我们提供了一个很好的思路，它试图从校验信息获取及实际校验过程两个方面来改善上述问题。先说说什么是哈希树，以最简单的二叉方式为例，如下图所示，设某文件共13个数据块，我们可以将其padding到16(2的整数次方)个块(注意，padding的空白块仅用于辅助校验，而无需当作数据进行实际传输)，每个块均对应一个SHA1校验值，然后再对它们进行反复的两两哈希，直到得出一个唯一的根哈希值为止(root hash, H0)，这一计算过程便构成了一棵二元的Merkle哈希树，树中最底层的叶子节点(H15~H30)对应着数据块的实际哈希值，而那些内部节点(H1~H14)我们则可以将其称为“路径哈希值”，它们构成了实际块哈希值与根哈希值H0之间的“校验路径”，比如，数据块8所对应的实际哈希值为H23，则有：SHA1(((SHA1(SHA1(H23,H24),H12),H6),H1)应该等于H0。当然，我们也可以进一步采用n元哈希树来进行上述校验过程，其过程是类似的。



采用Merkle哈希树校验方式能够极大地减小.torrent文件的尺寸，这是因为，一旦采用这种方式来校验数据块，那么便没有必要在.torrent文件中保存所有数据块的校验值了，其中仅需保存一个20字节的SHA1哈希值即可，即整个下载文件的根哈希值H0。

# 第4讲 安全体系结构（2）

1. **理解并实践Freebsd jail或者Jailkit方法；**

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，比较起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.

FreeBSDjail，一种操作系统层虚拟化技术，在FreeBSD操作系统中运作。利用这个技术，FreeBSD的系统管理者，可以创造出几个小型的软件系统，这些软件系统被称为监狱。

如何构建FreeBSD Jail：构建一个Jail，必备的几个参数有Jail的根目录，用户态程序，主机名以及IP地址，还有一个可选的dev。根据用户态程序的不同考虑可以将Jail做成完整的系统型Jail以及服务型Jail。

jailkit是一组实用程序，用于使用chroot（）和或特定命令将用户帐户限制到特定文件。

设置chroot shell、仅限于某些特定命令的shell或chroot监狱中的守护进程要容易得多，并且可以使用这些实用程序实现自动化。

1. **理解Chrome中的隔离和约束是如何实现的？**

Google Native Client (NaCl)

Native Client是Google在浏览器领域推出的一个开源技术

它允许在浏览器内执行原生的编译好的代码

好处：为Web提供更多的图形、音频以及其他功能；良好的可移植性；高性能；方便从桌面迁移；高安全性

外部沙箱: 使用系统调用介入限制功能

内部沙箱: 使用 x86 内存分段来隔离不同Web应用内存

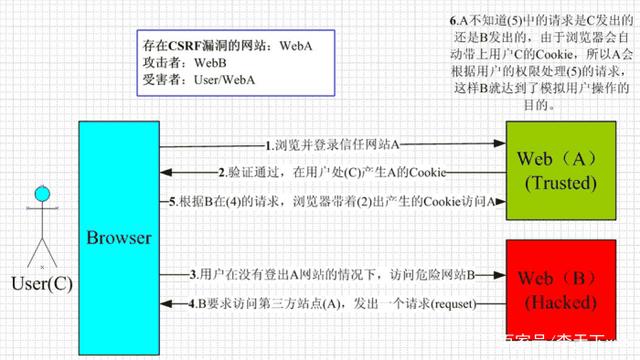
**三、查找相关参考文献，理解基于虚拟化的云系统中，隐蔽通道是如何实现的？**

虚拟机之间存在隐蔽通道，比如：基于从内存中读取数据所需时间的隐蔽通道

# 第6讲 Web安全

**Question 1: CSRF和XSS攻击的主要区别是什么？ 他们的名字都有“跨站点”。**

**CSRF的攻击原理**



用户是网站A的注册用户，且登录进去，于是网站A就给用户下发cookie。

从上图可以看出，要完成一次CSRF攻击，受害者必须满足两个必要的条件：

（1）登录受信任网站A，并在本地生成Cookie。（如果用户没有登录网站A，那么网站B在诱导的时候，请求网站A的api接口时，会提示你登录）

（2）在不登出A的情况下，访问危险网站B（其实是利用了网站A的漏洞）。

**CSRF如何防御**

方法一、Token 验证：（用的最多）

（1）服务器发送给客户端一个token；

（2）客户端提交的表单中带着这个token。

（3）如果这个 token 不合法，那么服务器拒绝这个请求。

方法二：隐藏令牌：

把 token 隐藏在 http 的 head头中。

方法二和方法一有点像，本质上没有太大区别，只是使用方式上有区别。

方法三、Referer 验证：

Referer 指的是页面请求来源。意思是，只接受本站的请求，服务器才做响应；如果不是，就拦截。

**XSS**

**XSS的攻击原理**

XSS攻击的核心原理是：不需要你做任何的登录认证，它会通过合法的操作（比如在url中输入、在评论框中输入），向你的页面注入脚本（可能是js、hmtl代码块等）。

最后导致的结果可能是：

盗用Cookie破坏页面的正常结构，插入广告等恶意内容D-doss攻击

**XSS攻击方式**

1、反射型

发出请求时，XSS代码出现在url中，作为输入提交到服务器端，服务器端解析后响应，XSS代码随响应内容一起传回给浏览器，最后浏览器解析执行XSS代码。这个过程像一次反射，所以叫反射型XSS。

2、存储型存

储型XSS和反射型XSS的差别在于，提交的代码会存储在服务器端（数据库、内存、文件系统等），下次请求时目标页面时不用再提交XSS代码。

XSS的防范措施（encode + 过滤）

**XSS防范措施**

1、编码：

对用户输入的数据进行

HTML Entity 编码。

把字符转换成 转义字符。

Encode的作用是将

$var

等一些字符进行转化，使得浏览器在最终输出结果上是一样的。

比如说这段代码：

<script>alert(1)</script>

若不进行任何处理，则浏览器会执行alert的js操作，实现XSS注入。

进行编码处理之后，L在浏览器中的显示结果就是

<script>alert(1)</script>

，实现了将$var作为纯文本进行输出，且不引起JavaScript的执行。

2、过滤：

移除用户输入的和事件相关的属性。如onerror可以自动触发攻击，还有onclick等。（总而言是，过滤掉一些不安全的内容）移除用户输入的Style节点、Script节点、Iframe节点。（尤其是Script节点，它可是支持跨域的呀，一定要移除）。

3、校正

避免直接对HTML Entity进行解码。使用DOM Parse转换，校正不配对的DOM标签。

DOM Parse

这个概念，它的作用是把文本解析成DOM结构（文档对象模型）。

比较常用的做法是，通过第一步的编码转成文本，然后第三步转成DOM对象，然后经过第二步的过滤。

还有一种简洁的答案：

首先是encode，如果是富文本，就白名单。

XSS和CSRF的区别

区别一：

CSRF：需要用户先登录网站A，获取 cookie。XSS：不需要登录。

区别二：（原理的区别）

CSRF：是利用网站A本身的漏洞，去请求网站A的api。XSS：是向网站 A 注入 JS代码，然后执行 JS 里的代码，篡改网站A的内容。