**1. 理解函数调用中堆栈工作原理**

（1）一个函数调用动作可分解为：零到多个PUSH指令（用于参数入栈），一个CALL指令，CALL指令内部其实还暗含了一个将返回地址（即CALL指令下一条指令的地址）压栈的动作。

（2）几乎所有本地编译器都会在每个函数体之前插入累死如下指令：PUSH EBP; MOVE EBP ESP.即在程序执行到一个函数的真正函数体是已经有一下数据顺序入栈：参数，返回地址，EBP。一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此时假设其占用4字节内存），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层EBP值，由于EBP中的地址处总是“上一层函数调用时的EBP值”，而在每一层函数调用中，都能通过当时的EBP值“向上（栈底方向）能获取返回地址，参数值，向下（栈顶方向）能获取函数局部变量值”，如此递归，直到到达栈底。

**2. 为什么要调用约定？比较集中典型的调用约定？**

所谓函数的调用约定其实就是处理函数参数入栈的顺序，多次理解stdcall和cdecl两种调用方式：

Stdcall是PASCAL约定，舱用于PASCAL函数和WIN 32 API中，函数采用从左到右的压栈方式，自己在退出时清空堆栈。

cdeck是c调用约定，按从右到左的顺序压参数入栈，由调用者吧参数弹出栈。传送参数的栈是由调用者维护的。每隔调用他的函数都包含清空堆栈的代码，所以产生的代码会稍大一些

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

（1）申明va-list类型变量；

（2）va-start将可变参数放到内存中；

（3）vsprintf格式化参数到buffer缓存空间；

（4）调用va-end清空可变参数列表。

Printf函数通过一个宏va-start把所有可变参数放到了由args只想的一块内存中，vsprintf将变参列表args中的变量按fmt中规定的格式保存到临时缓冲printbuffer中，最后调用puts函数将临时缓冲中的字符串数据打印到终端中。

**4. 阅读strncpy(),strncat()等函数代码，理解他们为什么是不安全的？**

（1）strncpy是不安全的且低效，若src长度小于n，则strncpy与strcpy效果不同，strncpy还会把dest剩下的部分全置为0，不安全。

（2）0strncat，为了目的地址操作不越界，徐指导目的地址的最大长度，strncat(char \*dst, char \*src, tsize count); strncat-s(char \*dst, tsize dstcount, char\* src, tsize maxcount);其中dstcount是为了复制过程中不会复制到超出dst上限而产生内存溢出，不安全。

**5. 理解return to libc的实现原理，理解ret2lic利用中栈的变化情况。**

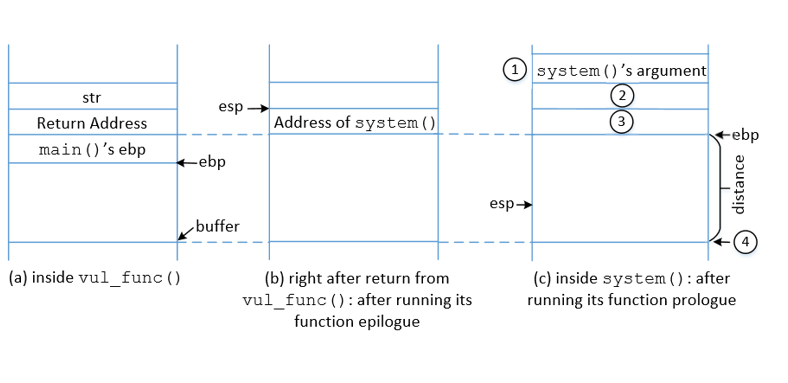
Return to libc控制流劫持攻击的一种方式，其实质仍然是利用了缓冲区溢出，劫持控制流指向libc中的系统函数，因此不用插入shellcode，对比普通的缓冲区溢出，ret2libc可以较好的抵抗栈不可执行保护。以实验中的ret2libc攻击为例，对栈变化进行说明：

（1）在原函数栈保存返回地址的位置插入系统中system函数地址

（2）system函数序言开始执行，首先将ebp压栈，然后将ebp改为当前esp的值

（3）在新的system函数栈中，当前ebp和esp所指向的位置为caller的ebp值，因此为了使得函数正常返回现在ebp+4的应当填充exit函数地址，又因为调用函数时第一个参数的位置是ebp+8，因此我们需要将/bin/sh放在现在ebp+8的位置。

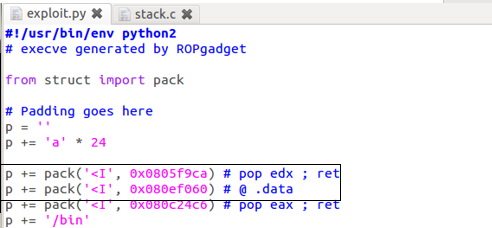
具体变化如下图所示



**6. 理解ROP的实现原理**

ROP控制流劫持攻击的一种方式，其实质仍然是利用了缓冲区溢出。在攻击中把libc看作以“ret”指令结尾的指令序列的库，使用指向这些序列的指针（和数据）填充栈，然后执行这些序列，产生需要的代码行为（图灵完备性），与。下面以实验中的ROP攻击栈填充的一部分为例，对ROP攻击进行说明。

如下图所示我们目标是存edx地址到寄存器edx中，与一般的缓冲区溢出类似首先我们需要对栈进行填充。（1）在保存函数返回地址的位置填入“pop edx；ret”这一指令片段在系统中的地址（2）执行完ret指令后，esp指向data地址（3）然后执行pop edx，将data地址存入edx中，此时esp指向栈中下一个操作的地址（4）最后执行ret指令，返回栈中保存的下一个操作（pop eax;ret）的地址。



**7. 理解控制流劫持的攻击方式和防御方式。**

（1）攻击方式：构造攻击载体，利用内存漏洞，篡改控制数据，绕过安全机制，最后完成控制流劫持攻击。

（2）防御方式：不可读保护。防止攻击者获取代码信息，隔离控制流相关数据，地址空间布局随机化ASLR和数据保护DEP。

**8. 说明dirtycow攻击原理，并说明dirtycow攻击后果。**

（1）需运行两个线程：线程1使用write（）写入映射内存；线程2丢弃映射内存的私有副本。

（2）低权限用户可利用该漏洞在linux上实现本地提权。影响范围为linux内核>=2.6.22（包括pc与android）

9. 调研并理解其他side-channel attacks方法。

旁道攻击是针对加密电子设备在运行过程中泄露的物理信息（如时间消耗、功率消耗或电磁辐射之类的侧信道信息）而对加密设备进行攻击的方法。其有效性远高于密码分析的数学方法，给密码设备带来了严重威胁。

**10. 说明Meltdown攻击原理。**

（1）恶意代码读取某块它感兴趣，但是没有权限访问的数据

（2）处理器读取这段高权限数据，将其放入高速缓存

（3）恶意代码随之读取对应的数据，如果发现访问时间长，则其将得知对应的数据在缓存里，否则就不在缓存里。

例如假设地址为2000的数据无法被恶意代码直接读取，但其值可能为1~5。此时，如果执行执行下列指令：“读取地址为5000+(地址为2000的数据)的数据”，则处理器可以读取的数据地址范围即为5001 ~ 5005。如果我们现在发起了对此地址的读取，然后再依次读取5001 ~ 5005地址的数据，而且发现5004地址的数据读取最快，则显然地址为2000的内存中存放的数据就是4了。

**11. 说明防御meltdown攻击的方法**

“内核页表隔离“（LPTI）软件将电脑内存和外部资源是为一个单一、连续的虚拟内存，LPTI禁止用户程序在运行时访问内存中的秘密数据。

**12.说明SPECTRE攻击原理**

Spectre攻击主要利用了指令的预测执行行为，切入方式是分支预测。

If(x<256){

Secret=array1[x];

Y=array[secret];

}

分支预测器期待下次的x小于256，攻击者使用“flush”指令清空处理器的告诉缓冲存储器。程序读取的任何指令须从主存中提取，最后系统地访问每个告诉缓冲存储器地址。由于secret值相同的内存地址是唯一已经从主存放入告诉缓冲存储器的地址，所以访问这一地址所需要时间低于其他地址访问时间，秘密由此泄露

**13. 分析BLP存在的隐蔽通道问题原理**

BLP通过隐蔽通道的非法信息流仍可能发生预计系统资源使用的通信通道，而这些通道一般不用于系统中的主体（进程）间的通信。

**14. 解释说明访问控制矩阵**

访问控制矩阵是利用矩阵的形式描述系统的访问控制的模型。由三元组(S,O,A)定义。

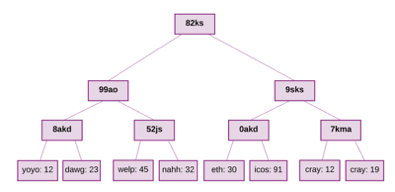
S是主体集合，行标对应主体；O是客体集合，列标对应客体。

A是访问矩阵，矩阵元素A[S,O]是主体S在O上实施的操作。

客体及其上实施的操作类型取决于应用系统本身的特点。

**15. 默克尔树模型的目标是从较少内存开销实现较快块数据完整性验证。以某种应用程序为例，分析它是如何实现这一目标的。**

以太坊区块头中的有三个树结构：状态根、 交易根、收据根。这些树结构就是以太坊中三个默克尔树的根，这里以状态根为例进行分析。我们知道以太坊的全局状态包括帐户地址和帐户状态之间的映射，这个映射存储在一个数据结构中，这种结构被称为Merkle Patricia树。Merkle Patricia树是一种由一组树状节点构成的二进制结构，它包括:（1）底层有大量的叶子节点，其中包含了潜在的数据；（2）一组中间节点，其中每个节点是其两个子节点的哈希；（3）一个单个的根节点，也是由它的两个子节点的哈希形成的，代表树的顶部。如下图为例。

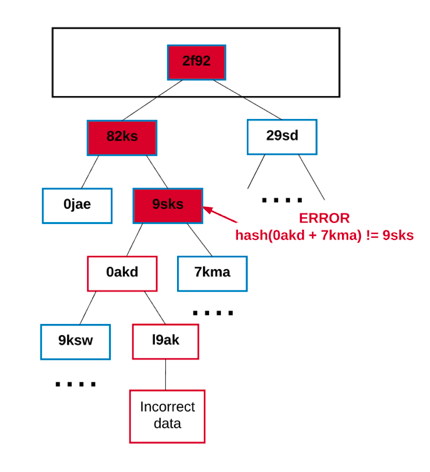


树的底部数据是通过把我们想要存储的数据分割成块后而生成的，然后将这些数据块分成几个桶，然后对每个桶的进行哈希迭代，值到剩下的哈希总数变为一个根哈希。此外，树需要存储在里面的每一个值的密钥。从树的根节点开始，密钥告诉你要遵循哪个子节点来获取相应的值，这些值存储在叶子节点中。在以太坊中，状态树的键值对是地址和相关帐户之间的映射，包括每个帐户的balance（余额）、nonce（交易量）、codeHash（合约代码哈希值）和storageRoot（storageRoot本身就是一棵树）。 同样的树结构也用于存储交易和收据，这样就形成了前文所述的以太坊区块头中的三个树形结构。

一个完整的节点需要下载完整的链，从元区块到当前的头部块，执行所有的交易也都包含其中。通常情况下，矿工储存完整的档案节点，因为他们必须这样做才能完成挖矿的过程。当然，也可以在不执行交易的情况下下载完整的节点。无论如何，任何完整的节点都包含整条链。

轻节点的概念与之相对，除非一个节点需要执行每个交易或查询历史数据，否则就没有必要存储整个链。这就是轻节点的意义所在。轻节点并不下载和存储完整链并执行所有的交易，而是只下载从元区块到当前头部区块的信息，而不执行任何交易或检索任何关联状态。因为轻节点可以访问包含三个树的区块头部哈希，所以仍然可以很容易地生成和接收关于交易、事件、余额等可验证的结果。这样做的原因是因为Merkle树中的哈希会向上传播ーー如果一个恶意用户试图将一个伪造的交易交换到Merkle树的底部，这种变化将导致上面节点的哈希变化，也将改变上面节点的哈希值。轻节点只需要下载与他想要验证交易块相关联的节点信息即可。

这样做的原因是如下图所示，因为Merkle树中的哈希会向上传播ーー如果一个恶意用户试图将一个伪造的交易交换到Merkle树的底部，这种变化将导致上面节点的哈希变化，也将改变上面节点的哈希值。总之，使用Merkle树的好处是，该结构的根节点依据树中存储的数据进行加密，因此根节点的哈希可以作为该数据的安全证明。由于区块头包括状态、交易和收据树的根哈希。因此任何节点都可以在不需要存储整个状态的情况下，验证以太坊的一小部分状态，而整个状态的大小可能是无限的。



**16. 理解并实践Freebsd jail或jailkit方法。**

Freebsd jail是一种操作系统虚拟化技术，在Freebsd操作系统中运作，利用这个技术，Freebsd的系统管理者可创出几个小型软件系统（监狱），它可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，还实现设备隔离，用户隔离使其更像一种虚拟机机制。

**17.理解chrome中的隔离和约束是如何实现的。**

Chrome启用站点隔离后，每个渲染器进程最多含一个站点的文档，即跨站点文档的所有导航会导致选项卡切换进程使用进程iframe将所有跨站点iframe放入与其父框架不同的进程中。站点隔离还包括一个称为CORB的功能，作为Fetch规范的一部分，可组织来自渲染器跨站点HTML，XML和JSON的响应。

**18.理解基于虚拟化云系统中，隐蔽通道是如何实现的？**

隐蔽通道主要有两种类型:存储通道和定时通道.如果一个进程直接或间接地写一个[存储单元](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%8D%95%E5%85%83/8727749),另一个进程直接或间接地读该存储单元,则称这种通道为隐蔽存储通道.如果一个进程通过调节它对[系统资源](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90)的使用,影响另外一个进程观察到的真实响应时间,实现一个进程向另一个进程传递信息,则称这种通道为隐蔽定时通道。