    Spectre以及meltdown漏洞是前段时间，十分热门的两个漏洞，它们之所以广受重视，是因为它们根据的是体系结构的设计漏洞，而非针对某个系统或者某个软件，因此它几乎可以遍及大多数近代的CPU。

这里主要有三个漏洞:

- Variant 1: bounds check bypass (CVE-2017-5753)【绕过边界检查】

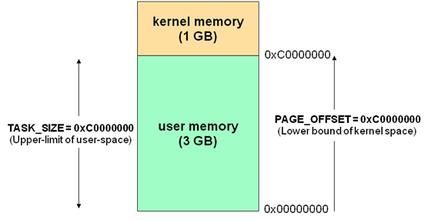
- Variant 2: branch target injection (CVE-2017-5715)【分支目标注入】

- Variant 3: rogue data cache load (CVE-2017-5754)【恶意数据缓存载入】

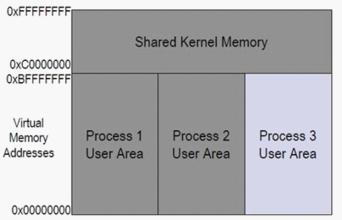
Spectre 主要利用前两个漏洞进行攻击，而meltdown则主要利用第三个漏洞进行攻击。

一、内存映射

        Linux和Windows的内存映射方法是不同的，在linux中，虚拟空间地址有4G，0~3G为用户空间，3~4G为内核空间

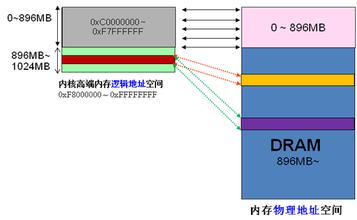


其中内核空间都相同，准确的讲是每个进程共享同一个内核空间

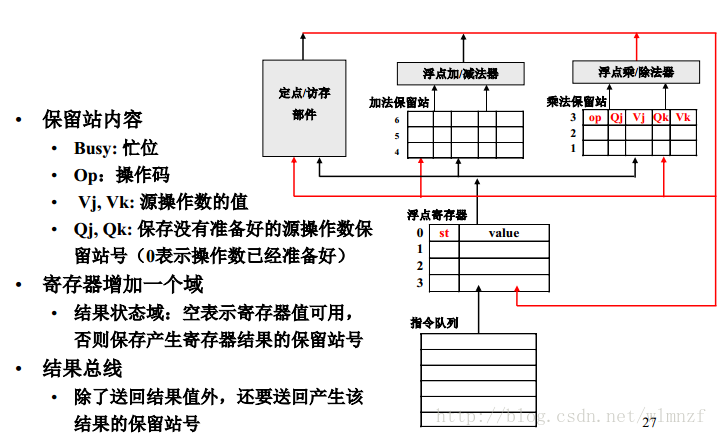


        Linux在启动时会初始化一个进程，然后通过fork()生成子进程，Linux的fork机制会把父进程的页表和堆栈等一模一样地复制一份，然后在运行时，子进程通过缺页异常等操作来改变用户空间，如果内核空间部分也改变了，则只修改初始化进程的内核空间，其它子进程访问该页时，再通过缺页中断将这部分内容从父进程更新过来。

        一般来讲，当进程进行系统调用进入内核态的时候，它应该能够访问整个地址空间的，但是在这里只能访问自己的1G的地址空间，于是我们需要通过地址的映射，来使他可以访问整个空间，内核空间分为：ZONE\_DMA（内存开始的16MB） 、ZONE\_NORMAL（16MB~896MB）、ZONE\_HIGHMEM（896MB ~ 结束）三个区域，其中0~896M是直接映射的，其余部分会进行非线性映射。



二、Tomasulo算法



        原始的Tomasulo算法是为了寄存器重命名以便消除指令的数据相关，数据一到就可以执行指令。

指令首先存在于指令队列之中。

指令一条一条地从队列中取出来，进行译码，然后放到对应的操作保留站中，比如加法指令放到加法保留站中。乘法指令放到乘除法保留站中。Vj和Vk中是用来存储源操作数的（已经就绪的源操作数值取自于浮点寄存器），若源操作数还没有准备好，则通过Qj和Qk指向操作数的保留站号，等到被指向的保留站中的操作计算出了结果，则结果可以通过结果总线传回到保留站中需要的指令，然后将其Qj和Qk中的值改为0（表示已经准备好）。

当Qj和Qk都为0时，那一条指令就可以送到乘/除器中执行。

三、乱序执行

        最早期 CPU都是顺序执行的，前一条指令未执行完毕，则后一条指令必须等待着，就像我们烧水，必须洗茶壶，烧水，洗茶杯，倒水必须按照顺序来做，但是事实上烧水和洗茶杯可以同时执行，其实有很多指令也是如此，调整它们的顺序可以加快程序执行的速度

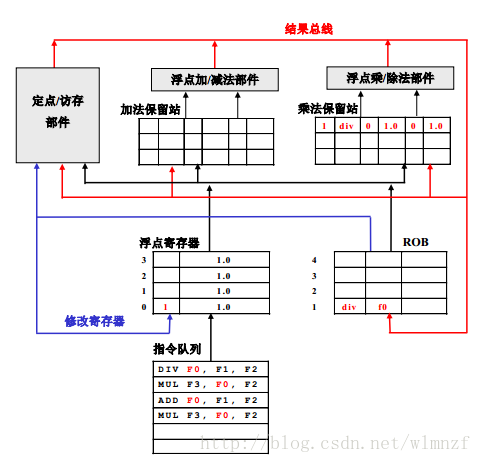
可能会造成乱序执行的原因：

1. 编译器为了优化而实现指令重排（静态调度）

2. CPU实现指令的多发射，以及并行执行，并为了优化实现指令的重排（动态调度）

        程序的乱序执行并不意味着在所有步骤中，指令的顺序都是混乱的，事实上，指令在发射时和提交时依然是顺序的，只是在执行的过程中会打乱顺序。

为了支持乱序执行



        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。

本次的漏洞就是利用了这一特性来实现的

四、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，我们加入了分支预测功能

静态分支

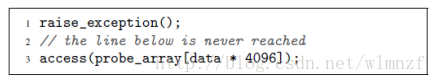
对于所有的跳转指令，我们都预测执行跳转或者执行不跳转，则称其为静态跳转

动态分支

动态分支将会通过历史跳转信息来预测下一次分支应该选择跳转还是选择不跳转。在intel设计中有一个称为BTB(Branch Target Buffer)的部件，当我们执行分支指令时，会将执行结果和分支指令地址记录在其中，当下次取址时，查询其中的记录，若存在，则根据历史执行记录进行预测是否跳转。

若不存在此记录，我们将会使用静态分支预测器，我们一般将向上跳转的分支指令看作循环，对于循环我们倾向于接受跳转，而对于向下跳转，我们倾向于不跳转。

五、Meltdown攻击



        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。

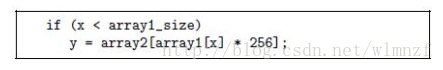


这段代码揭示了攻击的关键部分

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。

六、Spectre



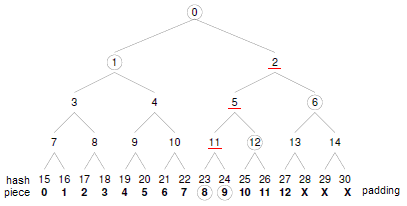
        Spectre主要利用了分支预测和乱序执行的漏洞实现的，如图所示的代码，看起来十分地正常，若x小于array1的长度的时候，循环顺利执行。但是我们假设这里有一个存储密码的变量地址为secret，并且令a=secret-array1，于是我们可以使用array1[a]来表示secret的值。当多次执行循环的时候，我们的x满足循环条件，则我们的分支预测模块会认为下一个循环也满足循环条件而去预执行这个循环，若此时我们将a的值赋值给x，则分支预测模块预测本次循环为执行（其实并不会执行），CPU会预执行这个循环体，然后将我们存储密码的secret值取出来，并将其作为地址去访问array2，但是最终发现循环不应该被执行，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）。

**Merkel Tree在BitTorren中的应用**

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，尤其是用于交换热门资源时；另一方面，为避免索引文件过大，人们在发布种子、制作索引文件时不得不采用增加其基本数据块长度(比如2MB)的方式来减少数据块总数，这将有可能对端对端数据交换性能造成一系列负面影响：因为块长越小，越能帮助我们及时地发现传输错误，试想在2MB块长的情况下，用户直到下载完整个数据块时才通过校验发现传输有误，然后不得不再次重传整个2MB块，这显然较块长为512KB时更加浪费带宽和时间。归根到底，导致上述麻烦的根本原因在于这么一个事实：“.torrent中包含了所有数据块的SHA1校验信息”。

有什么办法让我们既能获得较小的块长而又能减少索引文件长度？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。想象一下一个3、4GB的文件，其对应.torrent文件却只有100-200字节的样子？heh

下面，让我们来看看其数据块交换及校验的详细过程：

1. 以下载某大文件F为例，为启动BT交换，每一个client均必须从索引服务器中下载F所相应的.torrent文件，从而得到文件长度、数据块长及根哈希值H0等必要信息；
2. 任一client均可通过轮询Tracking服务器或者是DHT方式来定位其他参与交换F数据的peer，进而与之建立TCP连接进行P2P文件交换，其定位peer的方式基本上没有什么改变。
3. 当client从某一peer下载一个数据块时，它将事先向peer发出相应的校验查询请求，要求对方提供校验当前下载块所需的路径哈希值，这一过程可以很方便地通过扩展标准的Peer Wire Protocol实现。以上图为例，当client下载的是数据块8时，在下载开始之前，它将要求peer提供校验块8所需的全部路径哈希值：H24、H12、H6和H1；
4. 当client下载完某数据块之后，它将对其进行哈希树校验；仍以校验块8为例，client将首先计算出块8的哈希值H23'，然后再根据peer所提供的信息依次计算各层路径哈希值，直至求得一个唯一的根校验值H0'，如果H0'等于.torrent文件中的H0，则数据块下载无误，校验通过；反之则校验失败。
5. 一旦某数据块校验通过，与其哈希树校验过程相关的所有路径哈希值均可以作为cache缓存下来，而不管其究竟是从peer查询所得的，还是由client自身计算所得的；但值得注意的是，如果该数据块的校验未通过，则其所有相关的路径哈希值均不能被cache，而只能被废弃，因为此时我们无法保证peer所提供的路径哈希值的正确性(不能排除peer故意提供错误路径哈希值和哈希值传输出错的可能)。例如，所有与块8校验相关的路径哈希包括：H24、H12、H6、H1、H11、H5、H2，其中前面4个经查询peer所得，而后面3个则由client自己计算所得。块8验证的结果将决定这7个路径哈希是否可信及是否能被cache。
6. 当一定数量的路径哈希值被缓存之后，后继数据块的校验过程将被极大简化。此时我们可以直接利用校验路径上层次最低的已知路径哈希值来对数据块进行部分校验，而无需每次都校验至根哈希值H0，这主要是因为：某一路径哈希值被cache这一事实本身便能够证明其是可信的；例如，在块8校验完成之后，client无需进行额外的peer查询便可以直接对块9进行校验，因此它已经知道了H24的值，并且通过块8的校验过程验证了H24的正确性；而当client需要对块10进行校验时，它仅需向peer查询H26一个校验值即可，由于我们已经知道了H12值，因此对块10的校验仅需2次SHA1哈希即可：一次是计算块哈希H25，一次是计算SHA1(H25，H26)，并比较其等不等于H12。同理，当需要验证块12时，由于已知H6，client仅需查询2个路径哈希(H28和H14)和3次SHA1计算(H27、SHA1(H27,H28)和SHA1(H13,H14))即可。缓存的路径哈希值越多，则后继数据块校验越容易，速度也越快。
7. 为了进一步提高校验效率，可以考虑在.torrent文件上再做做文章：令其保存整个哈希树中最上面的若干层路径哈希值，而不仅仅只是一个根哈希值H0，底层的路径哈希值则仍然依靠询问peer或client自身计算所得，从而实现.torrent文件尺寸与校验效率的有效折衷。

最后再来看看上述算法的时空效率如何。假设某文件的总块数为M，将其padding至N个块(N＝2^p)，块长为K，不难看出：

1. 为了校验整个文件，共需缓存Sum(2^i)：i=0 to p，即2N-1个20字节的SHA1哈希值(包括根哈希值、所有路径哈希值和块哈希值)。以512KB为块长的一个4GB的文件为例，M=8000，则N=8192，p=13，所需存储空间为(2\*8192-1)\*20=327660 Bytes=319KB，其缓冲区较普通校验方式翻倍，但即便如此，校验所需的数据量仍完全可以接受，即使在最糟糕的情况下：完全不用cache、全部依赖peer查询来获得这些校验值，也无非总共增加了1个基本块不到的流量而已，而且这些流量还是通过P2P方式得到的。
2. 在无校验失败的理想情况下，校验整个文件共需M次块长为K的SHA1计算，和N-1次长度为20字节的SHA1运算。即，以哈希树方式校验块长512KB的4GB文件，共需进行8000次长度为512KB的SHA1计算，同时外加8191次长为2\*20字节的路径哈希计算，与普通校验方式相比，后者是哈希树额外引入的开销，但由于每次路径哈希的计算量非常小，其影响几可忽略。

综上所述，通过采用哈希树校验方式，我们可以将诸多校验所需哈希值的获取工作分散在P2P数据交换时捎带进行，而不是从.torrent文件中集中获取，从而有利于缓解索引服务器集中下载瓶颈，优化Peer to Peer数据传输性能；在实现上述目的的同时，哈希树校验机制仍能保证以P2P方式获取的校验信息的可靠性，在小块长的情况下，恶意peer几乎无法通过故意提供错误路径哈希值的方式来实现有效的服务拒绝攻击。采用这种方式，我们所需付出的额外代价主要包括几近1倍的校验缓存空间及其SHA1计算量的增长，但是经过简要分析不难看出其实际影响不甚明显，这对于换取较小的块长、提高大文件P2P交换效率而言是值得的。Merkle哈希树校验方式与分布式哈希表技术势必能够帮助BitTorrent协议进一步克服自身的非结构化缺陷，取得更大的应用发展。

RBAC

RBAC是什么？

[RBAC](http://www.sojson.com/tag_rbac.html)是基于角色的访问控制（Role-Based Access Control ）在[RBAC](http://www.sojson.com/tag_rbac.html)中，权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。这就极大地简化了权限的管理。这样管理都是层级相互依赖的，权限赋予给角色，而把角色又赋予用户，这样的权限设计很清楚，管理起来很方便。

**RBAC介绍。**

[RBAC](http://www.sojson.com/tag_rbac.html)认为授权实际上是Who 、What 、How 三元组之间的关系，也就是Who 对What 进行How 的操作，也就是“主体”对“客体”的操作。

**Who：是权限的拥有者或主体（如：User，Role）。**

**What：是操作或对象（operation，object）。**

**How：具体的权限（Privilege,正向授权与负向授权）。**

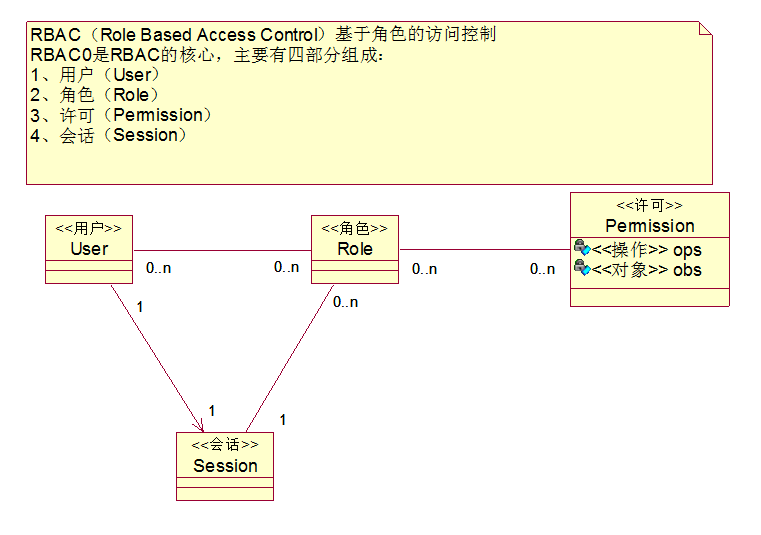
然后[RBAC](http://www.sojson.com/tag_rbac.html)又分为RBAC0、RBAC1、RBAC2、RBAC3 ，如果你不知道他们有什么区别，你可以百度百科：[百度百科-RBAC](http://baike.baidu.com/link?url=Tg3nxejvD2QVLLkjKa_4XaQoOWSPAVpR1FgHAG_gANcamtN2cYIm1r1irNw9VZ816FBrMEvdoYqwzixqdHd5e_) 估计你看不懂。还是看看我的简单介绍。

我这里结合我的见解，简单的描述下（去掉那么多的废话）。

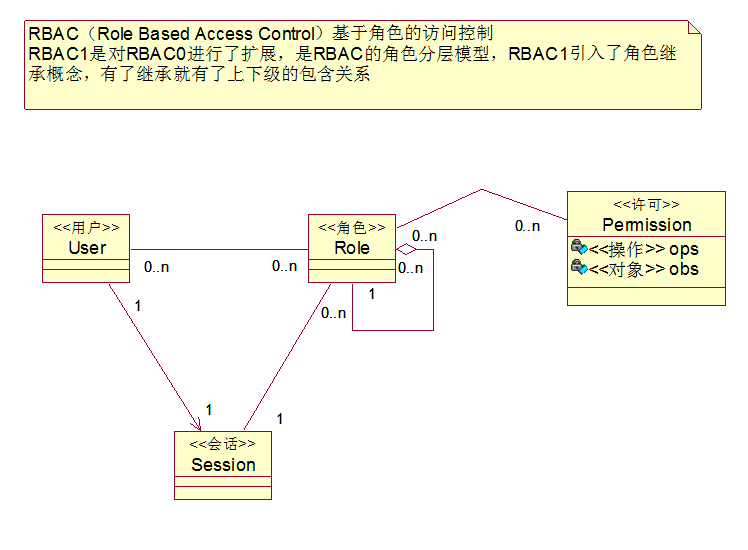
**RBAC0、RBAC1、RBAC2、RBAC3简单介绍。**

* **RBAC0：是RBAC的核心思想。**
* **RBAC1：是把RBAC的角色分层模型。**
* **RBAC2：增加了RBAC的约束模型。**
* **RBAC3：其实是RBAC2 + RBAC1。**

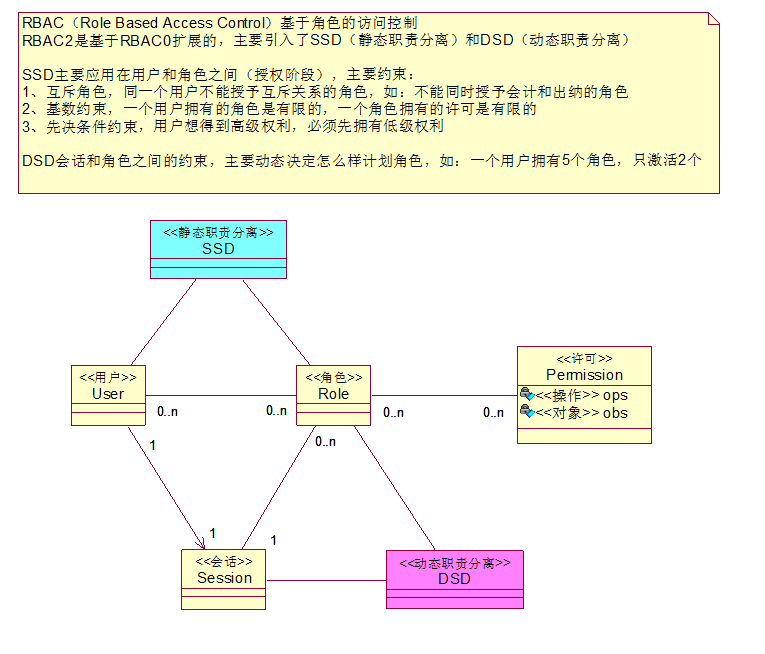
**RBAC0，RBAC的核心。**



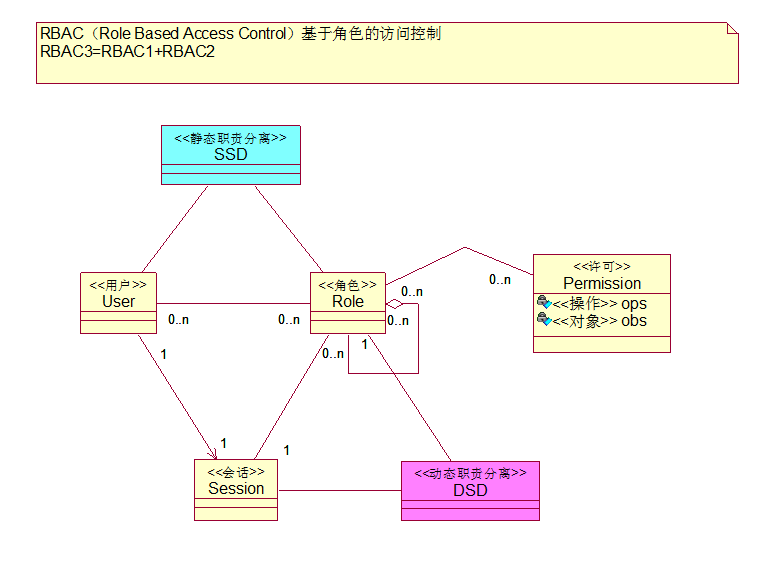
**RBAC1，基于角色的分层模型**

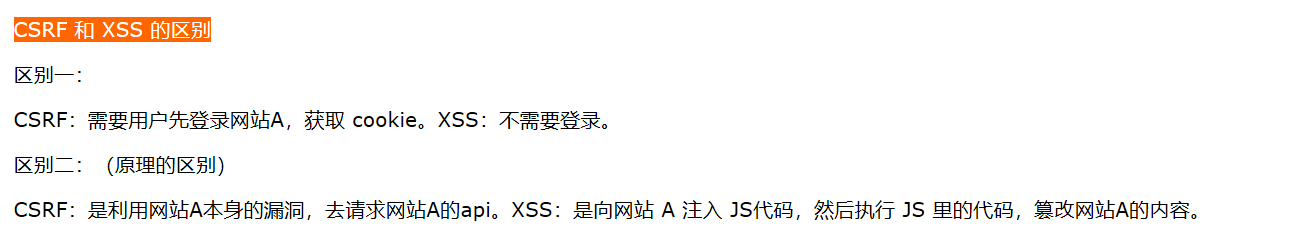


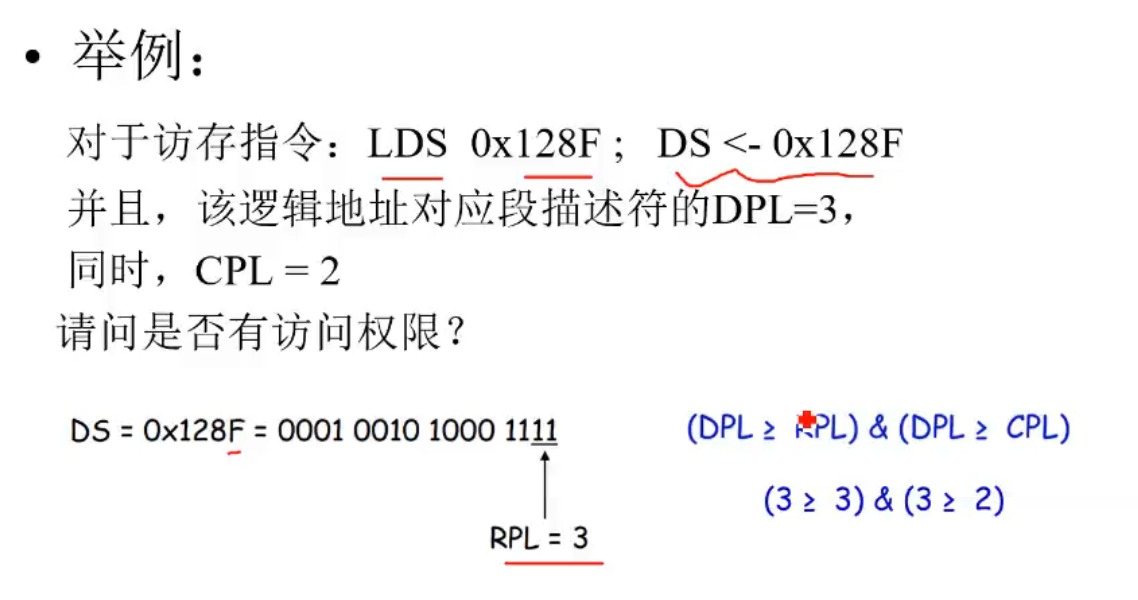
**RBAC2、是RBAC的约束模型。**



**RBAC3、就是RBAC1+RBAC2**







密钥分发中心（KDC）

密钥分发中心是一种运行在物理安全服务器上的服务，KDC维护着领域中所有安全主体账户信息数据库。

与每一个安全主体的其他信息一起，KDC存储了仅安全主体和KDC知道的加密密钥，这个密钥也称长效密钥（主密钥），用于在安全主体和KDC之间进行交换。

KDC是作为发起方和接收方共同信任的第三方，因为他维护者一个存储着该域中所有账户的账户数据库，也就是说，他知道属于每个账户的名称和派生于该账户密码的Master Key（主密钥）。而用于Alice和Bob相互认证的会话密钥就是由KDC分发的，下面详细讲解KDC分发会话密钥的过程。

分发会话密钥过程

1、首先客户端向KDC发送一个会话密钥申请。这个申请的内容可以简单概括为”我是某客户端，我需要个Session Key用于与某服务器通话“。

2、KDC在接收到这个请求的时候，生成一个会话密钥。为了保证这个会话密钥仅仅限于发送请求的客户端和它希望访问的服务器知道，KDC会为这个会话密钥生成两个副本，分别被客户端和服务器使用。然后从账户数据库中提取客户端和服务器的主密钥分别对这两个副本进行对称加密。对于服务器，与会话密钥一起被加密的还包含关于客户端的一些信息，以便对发起连接请求的客户端进行身份认证。

注意：KDC不是直接把这两个会话密钥副本分发客户端和服务器的，因为如果这样做，对于服务器来说会

出现下面两个问题。由于一个服务器会面对若干不同的客户端，而每个客户端都具有一个不同的Session

Key。那么服务器就会为所有的客户端维护这样一个会话密钥的列表，这样对服务器来说工作量就非常

大了。由于网络传输的不确定性，可能会出现这样一种情况：客户端很快获得会话密钥用于副本，并将

这个会话密钥作为凭据随同访问请求发送到服务器，但是用于服务器的会话密钥却还没有收到，并且很

有可能这个会话密钥永远也到不了服务器端，这样客户端将永远得不到认证。为了解决这个问题，

Kerberos将这两个被加密的副本一同发送给客户端，属于服务器的那份由客户端发送给服务器。因为

这两个会话密钥副本分别是由客户端和服务器端的主密钥加密的，所以不用担心安全问题。

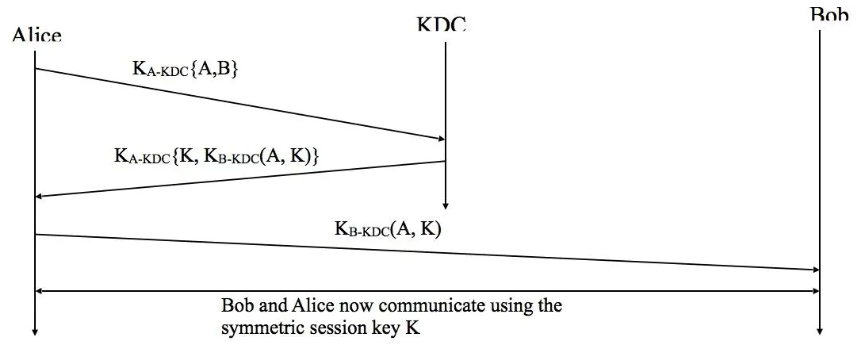
3、通过上面的过程，客户端实际上获得了两组信息：一个是通过自己主密钥加密的会话密钥；另一个是被Server的主密钥加密的数据包，包含会话密钥和关于自己的一些确认信息。在这个基础上，我们再来看看服务器是如何对客户端进行认证的。

4、客户端通过用自己的主密钥对KDC加密的会话密钥进行解密从而获得会话密钥，随后创建认证符（Authenticator，包括客户端信息和时间戳（Timestamp）），并用会话密钥对其加密。最后连同从KDC获得的、被服务器的主密钥加密过的数据包（客户端信息和会话密钥）一并发送到服务器端。我们把通过服务器的主密钥加密过的数据包称为服务票证（Session Ticket）。

5、当服务器接收到这两组数据后，先使用它自己的主密钥对服务票证进行解密，从而获得会话密钥。随后使用该会话密钥解密认证符，通过比较由客户端发送来的认证符中的客户端信息（Client Info）和服务票证中的客户端信息实现对客户端身份的验证。

双方进行了身份认证的同时也获得了会话密钥，那么双方可以进行会话了。

流程图如下：



客户端简称为Alice，服务端简称为Bob

Kerberos

kerberos 协议概述

前言

域环境：

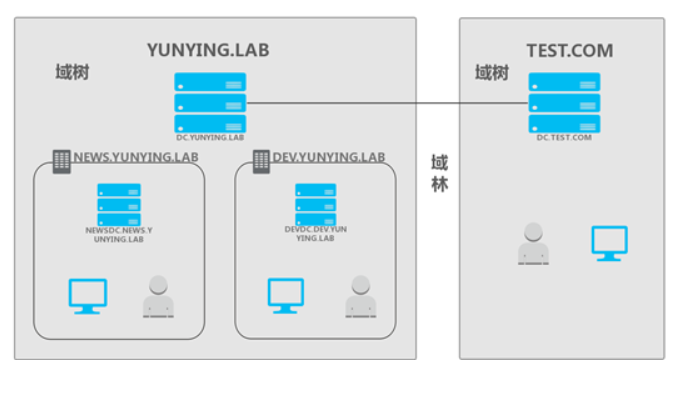
kerberos主要是在域环境做身份认证。什么是域环境呢？

域英文叫DOMAIN——域(Domain)是Windows网络中独立运行的单位，将网络中多台计算机逻辑上组织到一起，进行集中管理， 这种区别于工作组的逻辑环境叫做域。

域环境的作用：

MS以后不要将名称搞的这么专业,还是要考虑到中国的国情,早知道将"域"改成"中央"将"工作组"改成"自由市场",这样方便易懂,就不会有这么多的问题了。

在下图中 YUNYING.LAB 为其他两个域的根域，NEWS.YUNYING.LAB 和 DEV.YUNYING.LAB 均为 YUNYING.LAB 的子域，这三个域组成了一个域树。子域的概念可以理解为一个集团在不同业务上分公司，他们有业务重合的点并且都属于 YUNYING.LAB 这个根域，但又独立运作。同样 TEST.COM 也是一个单独的域树，两个域树 YUNYING.LAB 和 TEST.COM 组合起来被称为一个域林。



kerberos的认证过程

为什么需要kerberos：

kerberos目的是认证，既然认证就是辨别身份，那我输入用户名密码不就好了，为何要有Kerberos这样一个复杂的东西；举例来说，有A，B，C三个服务器，分别提供不同的服务，user要访问ABC都需要输入用户名密码，但是ABC没必要都存一份user的密码，所以就衍生出一个中央服务器D来专门存储用户名密码；如果user通过了D的认证，那就是合法的身份，就可以使用ABC任何一个服务，所以user需要告诉ABC它通过了D的认证。如何证明这个事情，以及信息在网络传输过程如何防止被截获篡改而假冒等等，解决这些问题就靠Kerberos。

kerberos的三个角色：

1.访问服务的用户

2.提供服务的server

3.KDC（ker DIstribution center）密钥分发中心

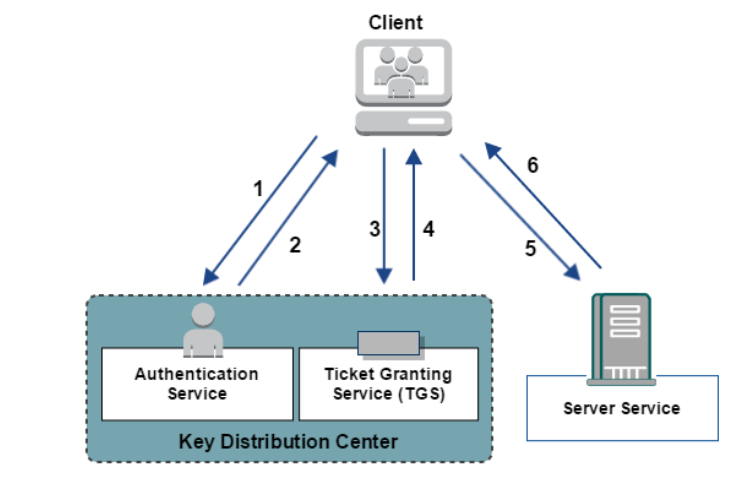
其中 KDC 服务默认会安装在一个域的域控中，而 Client 和 Server 为域内的用户或者是服务，如 HTTP 服务，SQL 服务。在 Kerberos 中 Client 是否有权限访问 Server 端的服务由 KDC 发放的票据来决定。

KDC 又分为两个部分：

Authentication Server（AS）：AS 的作用就是验证 Client 端的身份，验证通过就会给一张 TGT（Ticket Granting Ticket）票给 Client。

Ticket Granting Server（TGS）：TGS 的作用是通过 AS 发送给 Client 的票（TGT）换取访问 Server 端的票。

kerberos认证流程：



1>>> AS\_REQ:用户向KDA发起AS\_REQ，请求的凭据是用户hash加密的时间戳，内容为通过 Client 密码 Hash 加密的时间戳、ClientID、网络地址、加密类型等。

2>>> AS\_KDA:KDA对用户hash进行解密，验证成功后返回给用户由用户密码 HASH 加密的 sessionkey-as 和 TGT（由 KRBTGT HASH 加密的 sessionkey-as 和 TimeStamp 以及PAC，PAC里包含用户的sid，用户所在的组。）

3>>> TGS\_REQ:用户凭借TGT票据向KDC发起针对特定服务的TGS\_REQ请求。

Client 接收到了加密后的 Sessionkey-as 和 TGT 之后，用自身密码解密得到 Sessionkey-as，TGT 是由 KDC 密码加密，Client 无法解密。这时 Client 再用 Sessionkey-as 加密 TimeStamp 和 TGT 一起发送给 KDC 中的 TGS（TicketGranting Server）票据授权服务器换取能够访问 Server 的票据。

4>>> TGS\_REP:KDC使用krbtgt hash进行解密，如果正确，就返回用服务hash加密的TGS票据（这一步不管用户有没有对服务的访问权限，只要TGT是正确的，就会返回TGS票据）

TGS 收到用户发送过来的 TGT 和 Sessionkey-as 加密的 TimeStamp 之后，首先会检查自身是否存在 Client 所请求的服务。如果服务存在，则用 KRBTGT 密码解密 TGT。一般情况下 TGS 会检查 TGT 中的时间戳查看 TGT 是否过期，且原始地址是否和 TGT 中保存的地址相同。验证成功之后将用 sessionkey-as 加密的 sessionkey-tgs 和 Server 密码 HASH 加密的 Sessionkey-tgs 发送给 Client。

5>>> AP\_REQ:用户用获取的TGS票据去请求服务

6>>> AP\_REP:服务器使用自己的hash去解密TGS票据，如果解密正确，就拿解密得到的PAC去KDA询问用户有没有访问权限，KDC解密获取用户的sid以及用户所在的组，在根据服务的ACL，判断用户是否可以访问该服务。