前几天在给人解释Windows是如何通过Kerberos进行Authentication的时候，讲了半天也别把那位老兄讲明白，还差点把自己给绕进去。后来想想原因有以下两点：对于一个没有完全不了解Kerberos的人来说，Kerberos的整个Authentication过程确实不好理解——一会儿以这个Key进行加密、一会儿又要以另一个Key进行加密，确实很容易把人给弄晕；另一方面是我讲解方式有问题，一开始就从Kerberos的3个Sub-protocol全面讲述整个Authentication 过程，对于一个完全不了解Kerberos的人来说要求也忒高了点。为此，我花了一些时间写了这篇文章，尽量以由浅入深、层层深入的方式讲述我所理解的基于Kerberos的Windows Network Authentication，希望这篇文章能帮助那些对Kerberos不明就里的人带来一丝帮助。对于一些不对的地方，欢迎大家批评指正。

**一、 基本原理**

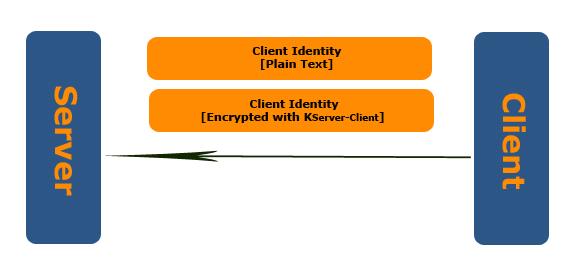
Authentication解决的是“如何证明某个人确确实实就是他或她所声称的那个人”的问题。对于如何进行Authentication，我们采用这样的方法：如果一个秘密（secret）仅仅存在于A和B，那么有个人对B声称自己就是A，B通过让A提供这个秘密来证明这个人就是他或她所声称的A。这个过程实际上涉及到3个重要的关于Authentication的方面：

* Secret如何表示。
* A如何向B提供Secret。
* B如何识别Secret。

基于这3个方面，我们把Kerberos Authentication进行最大限度的简化：整个过程涉及到Client和Server，他们之间的这个Secret我们用一个Key（**KServer-Client**）来表示。Client为了让Server对自己进行有效的认证，向对方提供如下两组信息：

* 代表Client自身Identity的信息，为了简便，它以明文的形式传递。
* 将Client的Identity使用**KServer-Client**作为Public Key、并采用对称加密算法进行加密。

由于**KServer-Client**仅仅被Client和Server知晓，所以被Client使用KServer-Client加密过的Client Identity只能被Client和Server解密。同理，Server接收到Client传送的这两组信息，先通过**KServer-Client**对后者进行解密，随后将机密的数据同前者进行比较，如果完全一样，则可以证明Client能过提供正确的**KServer-Client**，而这个世界上，仅仅只有真正的Client和自己知道**KServer-Client**，所以可以对方就是他所声称的那个人。

  
Keberos大体上就是按照这样的一个原理来进行Authentication的。但是Kerberos远比这个复杂，我将在后续的章节中不断地扩充这个过程，知道Kerberos真实的认证过程。为了使读者更加容易理解后续的部分，在这里我们先给出两个重要的概念：

* **Long-term Key/Master Key**：在Security的领域中，有的Key可能长期内保持不变，比如你在密码，可能几年都不曾改变，这样的Key、以及由此派生的Key被称为Long-term Key。对于Long-term Key的使用有这样的原则：被Long-term Key加密的数据不应该在网络上传输。原因很简单，一旦这些被Long-term Key加密的数据包被恶意的网络监听者截获，在原则上，只要有充足的时间，他是可以通过计算获得你用于加密的Long-term Key的——任何加密算法都不可能做到绝对保密。

在一般情况下，对于一个Account来说，密码往往仅仅限于该Account的所有者知晓，甚至对于任何Domain的Administrator，密码仍然应该是保密的。但是密码却又是证明身份的凭据，所以必须通过基于你密码的派生的信息来证明用户的真实身份，在这种情况下，一般将你的密码进行Hash运算得到一个Hash code, 我们一般管这样的Hash Code叫做Master Key。由于Hash Algorithm是不可逆的，同时保证密码和Master Key是一一对应的，这样既保证了你密码的保密性，有同时保证你的Master Key和密码本身在证明你身份的时候具有相同的效力。

* **Short-term Key/Session Key**：由于被Long-term Key加密的数据包不能用于网络传送，所以我们使用另一种Short-term Key来加密需要进行网络传输的数据。由于这种Key只在一段时间内有效，即使被加密的数据包被黑客截获，等他把Key计算出来的时候，这个Key早就已经过期了。

**二、引入Key Distribution: KServer-Client从何而来**

上面我们讨论了Kerberos Authentication的基本原理：通过让被认证的一方提供一个仅限于他和认证方知晓的Key来鉴定对方的真实身份。而被这个Key加密的数据包需要在Client和Server之间传送，所以这个Key不能是一个**Long-term Key**，而只可能是**Short-term Key**，这个可以仅仅在Client和Server的一个Session中有效，所以我们称这个Key为Client和Server之间的Session Key（**SServer-Client**）。

现在我们来讨论Client和Server如何得到这个**SServer-Client**。在这里我们要引入一个重要的角色：**Kerberos Distribution Center-KDC**。KDC在整个Kerberos Authentication中作为Client和Server共同信任的第三方起着重要的作用，而Kerberos的认证过程就是通过这3方协作完成。顺便说一下，Kerberos起源于希腊神话，是一支守护着冥界长着3个头颅的神犬，在keberos Authentication中，Kerberos的3个头颅代表中认证过程中涉及的3方：**Client、Server和KDC**。

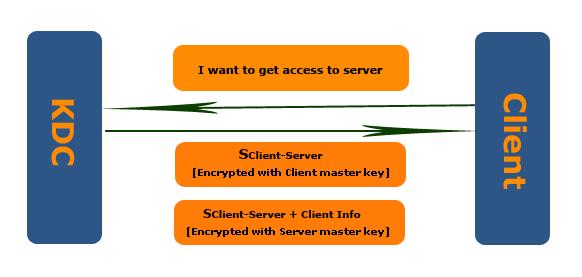
对于一个Windows Domain来说，**Domain Controller**扮演着KDC的角色。KDC维护着一个存储着该Domain中所有帐户的**Account Database**（一般地，这个Account Database由**AD**来维护），也就是说，他知道属于每个Account的名称和派生于该Account Password的**Master Key**。而用于Client和Server相互认证的**SServer-Client**就是有KDC分发。下面我们来看看KDC分发**SServer-Client**的过程。

通过下图我们可以看到KDC分发SServer-Client的简单的过程：首先Client向KDC发送一个对SServer-Client的申请。这个申请的内容可以简单概括为“**我是某个Client，我需要一个Session Key用于访问某个Server** ”。KDC在接收到这个请求的时候，生成一个Session Key，为了保证这个Session Key仅仅限于发送请求的Client和他希望访问的Server知晓，KDC会为这个Session Key生成两个Copy，分别被Client和Server使用。然后从Account database中提取Client和Server的Master Key分别对这两个Copy进行对称加密。对于后者，和Session Key一起被加密的还包含关于Client的一些信息。

KDC现在有了两个分别被Client和Server 的Master Key加密过的Session Key，这两个Session Key如何分别被Client和Server获得呢？也许你 马上会说，KDC直接将这两个加密过的包发送给Client和Server不就可以了吗，但是如果这样做，对于Server来说会出现下面 两个问题：

* 由于一个Server会面对若干不同的Client, 而每个Client都具有一个不同的Session Key。那么Server就会为所有的Client维护这样一个Session Key的列表，这样做对于Server来说是比较麻烦而低效的。
* 由于网络传输的不确定性，可能出现这样一种情况：Client很快获得Session Key，并将这个Session Key作为Credential随同访问请求发送到Server，但是用于Server的Session Key确还没有收到，并且很有可能承载这个Session Key的永远也到不了Server端，Client将永远得不到认证。

为了解决这个问题，Kerberos的做法很简单，将这两个被加密的Copy一并发送给Client，属于Server的那份由Client发送给Server。

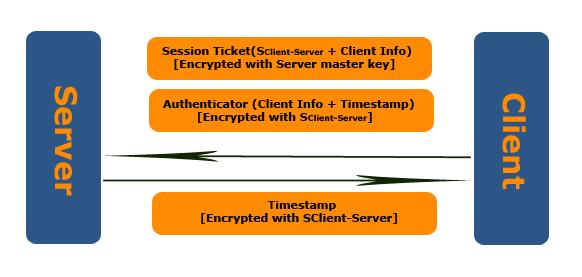
  
可能有人会问，KDC并没有真正去认证这个发送请求的Client是否真的就是那个他所声称的那个人，就把Session Key发送给他，会不会有什么问题？如果另一个人（比如Client B）声称自己是Client A，他同样会得到Client A和Server的Session Key，这会不会有什么问题？实际上不存在问题，因为Client B声称自己是Client A，KDC就会使用Client A的Password派生的Master Key对Session Key进行加密，所以真正知道Client A 的Password的一方才会通过解密获得Session Key。

**三、引入Authenticator - 为有效的证明自己提供证据**

通过上面的过程，Client实际上获得了两组信息：一个通过自己Master Key加密的Session Key，另一个被Sever的Master Key加密的数据包，包含Session Key和关于自己的一些确认信息。通过第一节，我们说只要通过一个双方知晓的Key就可以对对方进行有效的认证，但是在一个网络的环境中，这种简单的做法是具有安全漏洞，为此,Client需要提供更多的证明信息，我们把这种证明信息称为**Authenticator**，在Kerberos的Authenticator实际上就是**关于Client的一些信息**和当前时间的一个**Timestamp**（关于这个安全漏洞和Timestamp的作用，我将在后面解释）。

在这个基础上，我们再来看看Server如何对Client进行认证：Client通过**自己的Master Key**对KDC加密的Session Key进行解密从而获得**Session Key**，随后创建**Authenticator（Client Info + Timestamp）**并用**Session Key**对其加密。最后连同从KDC获得的、被**Server的Master Key**加密过的数据包（Client **Info + Session Key**）一并发送到Server端。我们把通过Server的Master Key加密过的数据包称为**Session Ticket**。

当Server接收到这两组数据后，先使用他**自己的Master Key**对Session Ticket进行解密，从而获得**Session Key**。随后使用该**Session Key**解密**Authenticator**，通过比较**Authenticator中的Client Info**和**Session Ticket中的Client Info**从而实现对Client的认证。

**  
为什么要使用Timestamp？**

到这里，很多人可能认为这样的认证过程天衣无缝：只有当Client提供正确的Session Key方能得到Server的认证。但是在现实环境中，这存在很大的安全漏洞。

我们试想这样的现象：Client向Server发送的数据包被某个恶意网络监听者截获，该监听者随后将数据包座位自己的Credential冒充该Client对Server进行访问，在这种情况下，依然可以很顺利地获得Server的成功认证。为了解决这个问题，Client在**Authenticator**中会加入一个当前时间的**Timestamp**。

在Server对Authenticator中的Client Info和Session Ticket中的Client Info进行比较之前，会先提取Authenticator中的**Timestamp**，并同**当前的时间**进行比较，如果他们之间的偏差超出一个可以**接受的时间范围（一般是5mins），**Server会直接拒绝该Client的请求。在这里需要知道的是，Server维护着一个列表，这个列表记录着在这个可接受的时间范围内所有进行认证的Client和认证的时间。对于时间偏差在这个可接受的范围中的Client，Server会从这个这个列表中获得**最近一个该Client的认证时间**，只有当**Authenticator中的Timestamp晚于通过一个Client的最近的认证时间**的情况下，Server采用进行后续的认证流程。

**Time Synchronization的重要性**

上述 基于Timestamp的认证机制只有在Client和Server端的时间保持同步的情况才有意义。所以保持Time Synchronization在整个认证过程中显得尤为重要。在一个Domain中，一般通过访问同一个**Time Service**获得当前时间的方式来实现时间的同步。

**双向认证（Mutual Authentication）**

Kerberos一个重要的优势在于它能够提供双向认证：**不但Server可以对Client 进行认证，Client也能对Server进行认证**。

具体过程是这样的，如果Client需要对他访问的Server进行认证，会在它向Server发送的Credential中设置一个是否需要认证的Flag。Server在对Client认证成功之后，会把Authenticator中的Timestamp提出出来，通过Session Key进行加密，当Client接收到并使用Session Key进行解密之后，如果确认**Timestamp**和原来的完全一致，那么他可以认定Server正式他试图访问的Server。

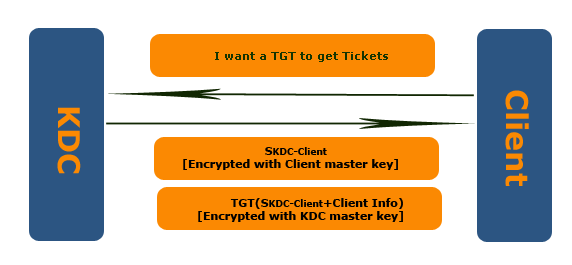
那么为什么Server不直接把通过Session Key进行加密的Authenticator原样发送给Client，而要把Timestamp提取出来加密发送给Client呢？原因在于防止恶意的监听者通过获取的Client发送的Authenticator冒充Server获得Client的认证。  
  
  
**四、引入Ticket Granting  Service**

通过上面的介绍，我们发现Kerberos实际上一个基于**Ticket**的认证方式。Client想要获取Server端的资源，先得通过Server的认证；而认证的先决条件是Client向Server提供从KDC获得的一个有**Server的Master Key**进行加密的**Session Ticket（Session Key + Client Info）**。可以这么说，Session Ticket是Client进入Server领域的一张门票。而这张门票必须从一个合法的Ticket颁发机构获得，这个颁发机构就是**Client和Server双方信任的KDC**， 同时这张Ticket具有超强的防伪标识：它是被Server的Master Key加密的。对Client来说， 获得Session Ticket是整个认证过程中最为关键的部分。

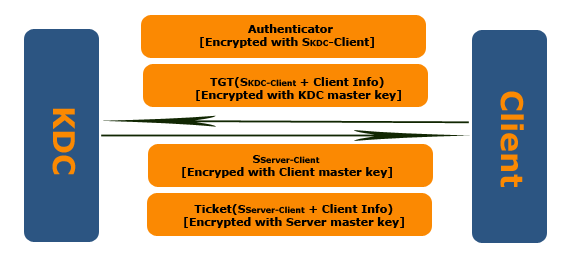
上面我们只是简单地从大体上说明了KDC向Client分发Ticket的过程，而真正在Kerberos中的Ticket Distribution要复杂一些。为了更好的说明整个Ticket Distribution的过程，我在这里做一个类比。现在的股事很火爆，上海基本上是全民炒股，我就举一个认股权证的例子。有的上市公司在股票配股、增发、基金扩募、股份减持等情况会向公众发行**认股权证**，认股权证的持有人可以凭借这个权证认购一定数量的该公司股票，认股权证是一种具有看涨期权的金融衍生产品。

而我们今天所讲的Client获得Ticket的过程也和通过认股权证购买股票的过程类似。如果我们把Client提供给Server进行认证的Ticket比作股票的话，那么Client在从KDC那边获得Ticket之前，需要先获得这个Ticket的认购权证，这个认购权证在Kerberos中被称为**TGT：Ticket Granting Ticket**，TGT的分发方仍然是KDC。

我们现在来看看Client是如何从KDC处获得TGT的：首先Client向KDC发起对TGT的申请，申请的内容大致可以这样表示：“**我需要一张TGT用以申请获取用以访问所有Server的Ticket**”。KDC在收到该申请请求后，生成一个用于该Client和KDC进行安全通信的**Session Key（SKDC-Client）**。为了保证该Session Key仅供该Client和自己使用，KDC使用**Client的Master Key**和**自己的Master Key**对生成的Session Key进行加密，从而获得两个加密的**SKDC-Client**的Copy。对于后者，随**SKDC-Client**一起被加密的还包含以后用于鉴定Client身份的关于Client的一些信息。最后KDC将这两份Copy一并发送给Client。这里有一点需要注意的是：为了免去KDC对于基于不同Client的Session Key进行维护的麻烦，就像Server不会保存**Session Key（SServer-Client）**一样，KDC也不会去保存这个Session Key（**SKDC-Client**），而选择完全靠Client自己提供的方式。

  
当Client收到KDC的两个加密数据包之后，先使用**自己的Master Key**对第一个Copy进行解密，从而获得KDC和Client的**Session Key（SKDC-Client）**，并把该Session 和TGT进行缓存。有了Session Key和TGT，Client自己的Master Key将不再需要，因为此后Client可以使用**SKDC-Client**向KDC申请用以访问每个Server的Ticket，相对于Client的Master Key这个Long-term Key，SKDC-Client是一个Short-term Key，安全保证得到更好的保障，这也是Kerberos多了这一步的关键所在。同时需要注意的是SKDC-Client是一个Session Key，他具有自己的生命周期，同时TGT和Session相互关联，当Session Key过期，TGT也就宣告失效，此后Client不得不重新向KDC申请新的TGT，KDC将会生成一个不同Session Key和与之关联的TGT。同时，由于Client Log off也导致SKDC-Client的失效，所以SKDC-Client又被称为**Logon Session Key**。

接下来，我们看看Client如何使用TGT来从KDC获得基于某个Server的Ticket。在这里我要强调一下，Ticket是基于某个具体的Server的，而TGT则是和具体的Server无关的，Client可以使用一个TGT从KDC获得基于不同Server的Ticket。我们言归正传，Client在获得自己和KDC的**Session Key（SKDC-Client）**之后，生成自己的Authenticator以及所要访问的Server名称的并使用**SKDC-Client**进行加密。随后连同TGT一并发送给KDC。KDC使用**自己的Master Key**对TGT进行解密，提取Client Info和**Session Key（SKDC-Client）**，然后使用这个**SKDC-Client**解密Authenticator获得Client Info，对两个Client Info进行比较进而验证对方的真实身份。验证成功，生成一份基于Client所要访问的Server的Ticket给Client，这个过程就是我们第二节中介绍的一样了。

**  
五、Kerberos的3个Sub-protocol：整个Authentication**

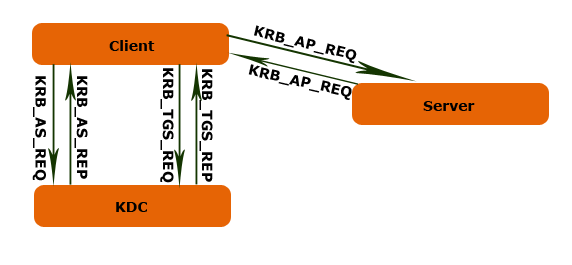
通过以上的介绍，我们基本上了解了整个Kerberos authentication的整个流程：整个流程大体上包含以下3个子过程：

1. Client向KDC申请TGT（Ticket Granting Ticket）。
2. Client通过获得TGT向DKC申请用于访问Server的Ticket。
3. Client最终向为了Server对自己的认证向其提交Ticket。

不过上面的介绍离真正的Kerberos Authentication还是有一点出入。Kerberos整个认证过程通过3个sub-protocol来完成。这个3个Sub-Protocol分别完成上面列出的3个子过程。这3个sub-protocol分别为：

1. Authentication Service Exchange
2. Ticket Granting Service Exchange
3. Client/Server Exchange

下图简单展示了完成这个3个Sub-protocol所进行Message Exchange。

**  
1． Authentication Service Exchange**

通过这个Sub-protocol，KDC（确切地说是KDC中的Authentication Service）实现对Client身份的确认，并颁发给该Client一个TGT。具体过程如下：

Client向KDC的Authentication Service发送Authentication Service Request（**KRB\_AS\_REQ**）, 为了确保KRB\_AS\_REQ仅限于自己和KDC知道，Client使用自己的Master Key对KRB\_AS\_REQ的主体部分进行加密（KDC可以通过Domain 的Account Database获得该Client的Master Key）。KRB\_AS\_REQ的大体包含以下的内容：

* Pre-authentication data：包含用以证明自己身份的信息。说白了，就是证明自己知道自己声称的那个account的Password。一般地，它的内容是一个被Client的Master key加密过的Timestamp。
* Client name & realm: 简单地说就是Domain name\Client
* Server Name：注意这里的Server Name并不是Client真正要访问的Server的名称，而我们也说了TGT是和Server无关的（Client只能使用Ticket，而不是TGT去访问Server）。这里的Server Name实际上是**KDC的Ticket Granting Service的Server Name**。

AS（Authentication Service）通过它接收到的KRB\_AS\_REQ验证发送方的是否是在Client name & realm中声称的那个人，也就是说要验证发送放是否知道Client的Password。所以AS只需从Account Database中提取Client对应的Master Key对Pre-authentication data进行解密，如果是一个合法的Timestamp，则可以证明发送放提供的是正确无误的密码。验证通过之后，AS将一份Authentication Service Response（KRB\_AS\_REP）发送给Client。KRB\_AS\_REQ主要包含两个部分：本Client的Master Key加密过的Session Key（SKDC-Client：Logon Session Key）和被自己（KDC）加密的TGT。而TGT大体又包含以下的内容：

* Session Key: SKDC-Client：Logon Session Key
* Client name & realm: 简单地说就是Domain name\Client
* End time: TGT到期的时间。

Client通过自己的Master Key对第一部分解密获得Session Key（SKDC-Client：Logon Session Key）之后，携带着TGT便可以进入下一步：TGS（Ticket Granting Service）Exchange。

**2． TGS（Ticket Granting Service）Exchange**

TGS（Ticket Granting Service）Exchange通过Client向KDC中的TGS（Ticket Granting Service）发送Ticket Granting Service Request（**KRB\_TGS\_REQ**）开始。KRB\_TGS\_REQ大体包含以下的内容：

* TGT：Client通过AS Exchange获得的Ticket Granting Ticket，TGT被KDC的Master Key进行加密。
* Authenticator：用以证明当初TGT的拥有者是否就是自己，所以它必须以TGT的办法方和自己的Session Key（SKDC-Client：Logon Session Key）来进行加密。
* Client name & realm: 简单地说就是Domain name\Client。
* Server name & realm: 简单地说就是Domain name\Server，这回是Client试图访问的那个Server。

TGS收到KRB\_TGS\_REQ在发给Client真正的Ticket之前，先得整个Client提供的那个TGT是否是AS颁发给它的。于是它不得不通过Client提供的Authenticator来证明。但是Authentication是通过**Logon Session Key（SKDC-Client）**进行加密的，而自己并没有保存这个Session Key。所以TGS先得通过自己的Master Key对Client提供的TGT进行解密，从而获得这个Logon Session Key（SKDC-Client），再通过这个**Logon Session Key（SKDC-Client）**解密Authenticator进行验证。验证通过向对方发送Ticket Granting Service Response（KRB\_TGS\_REP）。这个KRB\_TGS\_REP有两部分组成：使用**Logon Session Key（SKDC-Client）**加密过用于Client和Server的**Session Key（SServer-Client）**和使用**Server的Master Key**进行加密的Ticket。该Ticket大体包含以下一些内容：

* Session Key：SServer-Client。
* Client name & realm: 简单地说就是Domain name\Client。
* End time: Ticket的到期时间。

Client收到KRB\_TGS\_REP，使用**Logon Session Key（SKDC-Client）**解密第一部分后获得**Session Key（SServer-Client）**。有了Session Key和Ticket，Client就可以之间和Server进行交互，而无须在通过KDC作中间人了。所以我们说Kerberos是一种高效的认证方式，它可以直接通过Client和Server双方来完成，不像Windows NT 4下的NTLM认证方式，每次认证都要通过一个双方信任的第3方来完成。

我们现在来看看 Client如果使用Ticket和Server怎样进行交互的，这个阶段通过我们的第3个Sub-protocol来完成：**CS（Client/Server ）Exchange**。

**3． CS（Client/Server ）Exchange**

这个已经在本文的第二节中已经介绍过，对于重复发内容就不再累赘了。Client通过TGS Exchange获得Client和Server的**Session Key（SServer-Client）**，随后创建用于证明自己就是Ticket的真正所有者的Authenticator，并使用**Session Key（SServer-Client）**进行加密。最后将这个被加密过的Authenticator和Ticket作为Application Service Request（KRB\_AP\_REQ）发送给Server。除了上述两项内容之外，KRB\_AP\_REQ还包含一个Flag用于表示Client是否需要进行双向验证（Mutual Authentication）。

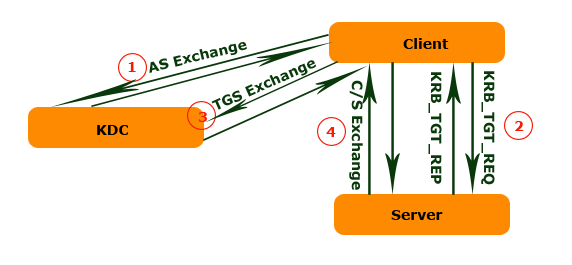
Server接收到KRB\_AP\_REQ之后，通过自己的Master Key解密Ticket，从而获得Session Key（SServer-Client）。通过Session Key（SServer-Client）解密Authenticator，进而验证对方的身份。验证成功，让Client访问需要访问的资源，否则直接拒绝对方的请求。

对于需要进行双向验证，Server从Authenticator提取Timestamp，使用Session Key（SServer-Client）进行加密，并将其发送给Client用于Client验证Server的身份。  
  
  
**六、User2User Sub-Protocol：有效地保障Server的安全**

通过3个Sub-protocol的介绍，我们可以全面地掌握整个Kerberos的认证过程。实际上，在Windows 2000时代，基于Kerberos的Windows Authentication就是按照这样的工作流程来进行的。但是我在上面一节结束的时候也说了，基于3个Sub-protocol的Kerberos作为一种Network Authentication是具有它自己的局限和安全隐患的。我在整篇文章一直在强调这样的一个原则：**以某个Entity的Long-term Key加密的数据不应该在网络中传递**。原因很简单，所有的加密[**算法**](http://lib.csdn.net/base/datastructure)都不能保证100%的安全，对加密的数据进行解密只是一个时间的过程，最大限度地提供安全保障的做法就是：**使用一个Short-term key（Session Key）代替Long-term Key对数据进行加密，使得恶意用户对其解密获得加密的Key时，该Key早已失效**。但是对于3个Sub-Protocol的C/S Exchange，Client携带的Ticket却是被**Server Master Key**进行加密的，这显现不符合我们提出的原则，降低Server的安全系数。

所以我们必须寻求一种解决方案来解决上面的问题。这个解决方案很明显：就是采用一个Short-term的Session Key，而不是Server Master Key对Ticket进行加密。这就是我们今天要介绍的Kerberos的第4个Sub-protocol：**User2User Protocol**。我们知道，既然是Session Key，仅必然涉及到两方，而在Kerberos整个认证过程涉及到3方：Client、Server和KDC，所以用于加密Ticket的只可能是Server和KDC之间的**Session Key（SKDC-Server）。**

我们知道Client通过在AS Exchange阶段获得的TGT从KDC那么获得访问Server的Ticket。原来的Ticket是通过**Server的Master Key**进行加密的，而这个Master Key可以通过Account Database获得。但是现在KDC需要使用Server和KDC之间的**SKDC-Server**进行加密，而KDC是不会维护这个Session Key，所以**这个Session Key只能靠申请Ticket的Client提供**。所以在AS Exchange和TGS Exchange之间，Client还得对Server进行请求已获得Server和KDC之间的Session Key（**SKDC-Server**）。而对于Server来说，它可以像Client一样通过**AS Exchange**获得他和KDC之间的Session Key（**SKDC-Server**）和一个封装了这个Session Key并被**KDC的Master Key进行加密的TGT**，一旦获得这个TGT，Server会缓存它，以待Client对它的请求。我们现在来详细地讨论这一过程。

  
上图基本上翻译了基于User2User的认证过程，这个过程由4个步骤组成。我们发现较之我在上面一节介绍的基于传统3个Sub-protocol的认证过程，这次对了第2部。我们从头到尾简单地过一遍：

1. AS Exchange：Client通过此过程获得了属于自己的TGT，有了此TGT，Client可凭此向KDC申请用于访问某个Server的Ticket。
2. 这一步的主要任务是获得封装了Server和KDC的Session Key（SKDC-Server）的属于Server的TGT。如果该TGT存在于Server的缓存中，则Server会直接将其返回给Client。否则通过AS Exchange从KDC获取。
3. TGS Exchange：Client通过向KDC提供自己的TGT，Server的TGT以及Authenticator向KDC申请用于访问Server的Ticket。KDC使用先用自己的Master Key解密Client的TGT获得SKDC-Client，通过SKDC-Client解密Authenticator验证发送者是否是TGT的真正拥有者，验证通过再用自己的Master Key解密Server的TGT获得KDC和Server 的Session Key（SKDC-Server），并用该Session Key加密Ticket返回给Client。
4. C/S Exchange：Client携带者通过KDC和Server 的Session Key（SKDC-Server）进行加密的Ticket和通过Client和Server的Session Key（SServer-Client）的Authenticator访问Server，Server通过SKDC-Server解密Ticket获得SServer-Client，通过SServer-Client解密Authenticator实现对Client的验证。

这就是整个过程。

**七、Kerberos的优点**

分析整个Kerberos的认证过程之后，我们来总结一下Kerberos都有哪些优点：

**1．较高的Performance**

虽然我们一再地说Kerberos是一个涉及到3方的认证过程：Client、Server、KDC。但是一旦Client获得用过访问某个Server的Ticket，该Server就能根据这个Ticket实现对Client的验证，而无须KDC的再次参与。和传统的基于Windows NT 4.0的每个完全依赖Trusted Third Party的NTLM比较，具有较大的性能提升。

**2．实现了双向验证（Mutual Authentication）**

传统的NTLM认证基于这样一个前提：Client访问的远程的Service是可信的、无需对于进行验证，所以NTLM不曾提供双向验证的功能。这显然有点理想主义，为此Kerberos弥补了这个不足：Client在访问Server的资源之前，可以要求对Server的身份执行认证。

**3．对Delegation的支持**

Impersonation和Delegation是一个分布式环境中两个重要的功能。Impersonation允许Server在本地使用Logon 的Account执行某些操作，Delegation需用Server将logon的Account带入到另过一个Context执行相应的操作。NTLM仅对Impersonation提供支持，而Kerberos通过一种双向的、可传递的（Mutual 、Transitive）信任模式实现了对Delegation的支持。

**4．互操作性（Interoperability）**

Kerberos最初由MIT首创，现在已经成为一行被广泛接受的标准。所以对于不同的平台可以进行广泛的互操作。