http://www.sqlservercentral.com/articles/Security/65169/
Printed 2008/12/11 06:13PM

Configuring Kerberos Authentication

By Brian Kelley, 2008/12/05

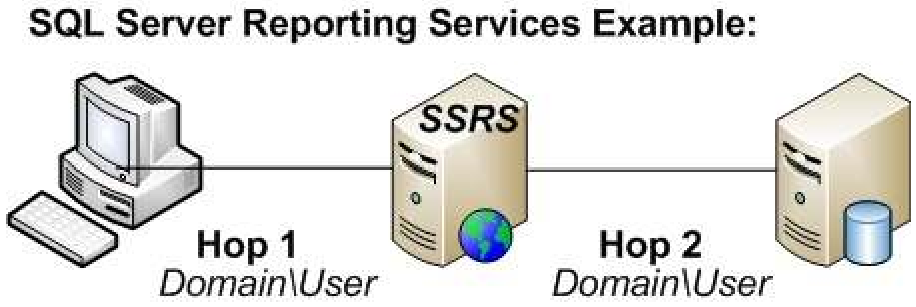
In my experience, configuring a SQL Server for Kerberos authentication, especially a SQL Server named instance, can be one of the most confusing things to do for a DBA or system administrator the first time around. The reason it can be so confusing is there are several "moving parts" that must all be in sync for Kerberos authentication to work. And what can make things all the more confusing is that in general, if we don't touch a thing, people and applications can connect to our database servers but as soon as we start down the road of configuring Kerberos authentication, they suddenly can't. And it can be rather frustrating to figure out why. In this article we'll look at both the hows and the whys.

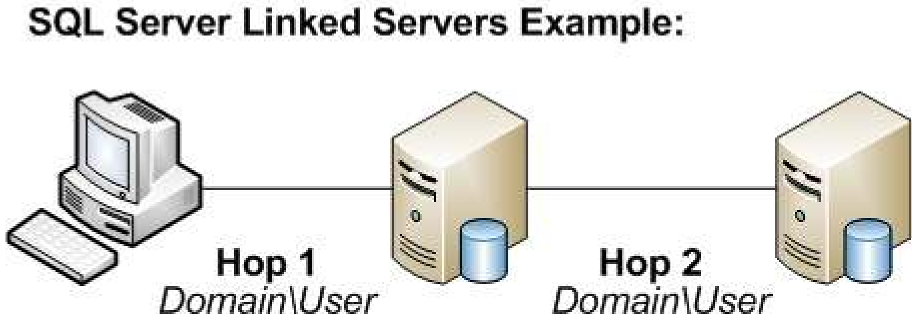
If I Don't Do Anything, Why Does it Usually Work?

When it comes to authenticating a login (checking to see if you are who you say you are), SQL Server only does authentication when the login is a SQL Server based one. I've chosen my words carefully here, because it is important to understand that when it's a Windows-based login, SQL Server passes off the authentication to an operating system component, the Windows Security Support Provider Interface (SSPI). That's why when you have Kerberos authentication errors, you usually get some message about SSPI context. Basically, SQL Server realizes it's a Windows login, gets the information it'll need to pass on so SSPI can do it's checks, and then it waits to see what SSPI says. If SSPI says the login is good, SQL Server allows the login to complete the connection. If SSPI says the login is bad, SQL Server rejects the login and returns whatever error information SSPI provides. Now, there is one exception to SQL Server farming out Windows authentication to SSPI, but that occurs in Named Pipes and so we won't get into it because hopefully you're not using Named Pipes as your protocol.

Once we understand that SQL Server is handing off responsibility for authentication to SSPI, it's time to understand what SSPI is going to do. SSPI is going to first try and authenticate using Kerberos. This is the preferred protocol for Windows 2000 and above. In order to do this, there needs to be a Service Principal Name (SPN) in place. We'll talk more about that later. If there's no SPN, Kerberos can't happen. If Kerberos can't happen whether due to no SPN or another reason (across forests with no forest level trust), SSPI will drop back to the old security protocol, NT LAN Manager, or NTLM. So if we don't do anything, authentication will drop back to NTLM and everything tends to work. That is, until we have to do multiple "hops," like through SQL Server Reporting Services set up on a separate server or when we want to do Windows authentication across a linked server connection (see Figure 1).

Figure 1:





In Figure 1, the same set of credentials (Domain\User) is being passed from the client to a server and then from that server to a second server. Each time the credentials are passed, we call that a hop. Since we're not changing the credentials (for instance, we're not going to a second Windows account, such as a service account, or a SQL Server login, we say that there have been two hops, or what we call that a double hop situation. NTLM doesn't permit double hop situations (or triple or quadruple…); It is prevented by design. So in either of these particular scenarios, if we don't have Kerberos authentication set up, we can't make the second hop. We'll see errors logging in attributed to **login (null)** or **NT AUTHORITY\ANONYMOUS LOGON**. By default, Kerberos authentication only permits a single hop, but using a feature called Kerberos delegation, multiple hops can be configured and these double hop scenarios can be allowed. While Kerberos delegation is beyond the scope of this article, it is important to note that Kerberos delegation cannot happen without Kerberos authentication, and that's how DBAs usually get pulled into the fray.

What's So Bad About NTLM?

In general, NTLM (or at least, the revised versions) do a good job of authenticating the user and basically being secure. However, NTLM suffers from the following drawbacks:

- It is susceptible to "replay" attacks.
- It assumes the server is trustworthy.
- It requires more authentication traffic than Kerberos.
- It doesn't provide for a means of going past that first hop.

Let's look at each one of these to understand why they are drawbacks, starting with a replay attack. A replay attack is when an attacker is able to capture network traffic and re-use it. For instance, imagine I'm logging on to your SQL Server. An attacker has a packet sniffer and is able to capture that logon sequence. If, at a later point, that attacker could put that traffic back on the network and it work, that would be a replay attack. The classic example given is an attacker captures a bank transaction for some amount of money. Let's say you pay Mr. Attacker US$500 for services rendered. If the attacker can capture the network traffic and replay it multiple times, the bank will deduct US$500 from your account each time and deposit it into his. To the bank, the repeated transactions looked legitimate (although admittedly, with everyone worried about fraud nowadays, we would hope this kind of thing gets flagged and checked out). If this is the case, then the protocol for that transaction we're using is to blame because it provided us no protection from such an attack. Such is the case with NTLM. It provides no protection. Kerberos, on the other hand, includes a time stamp of when the network traffic was sent. If you're outside the window of the acceptable time range (by default this is 5 minutes), Kerberos rejects that network traffic. So in the case above, imagine if the bank put a timestamp on the transaction and had an acceptable time range within 10 seconds. If Mr. Attacker tried to replay the transaction after that 10 second window was up, the bank would know something was going on.

The second drawback with NTLM is that the server isn't verified. The client connects to MySQLServer. Or at least, it thinks it is connecting to MySQLServer. The NTLM protocol may have the ability to validate that Domain\User is connecting, but it doesn't allow Domain\User to verify that he or she is really talking to MySQLServer. This is where the Service Principal Name (SPN) comes into play. When the client attempts to connect via Kerberos, the SPN for the service being connected to is checked. In a Windows 2000 or higher domain, the SPN is stored within Active Directory, and the Active Directory domain controller is trusted by the client. Therefore, if the service, such as a SQL Server service, checks out based on the SPN the client finds for that service within Active Directory, it knows that it can trust the server is truly MySQLServer.

The third drawback is the amount of authentication traffic used by NTLM versus Kerberos. In NTLM, every time authentication happens, a check has to be made back to a domain controller (DC). With Kerberos, tickets are issued to both the client and the server containing the information each needs to validate the other. Therefore, the client and the server only have to check in with a domain controller once during the lifespan of those tickets (default is 600 minutes or 10 hours) to get the tickets in the first place. After that, they both have the information they need without checking back with a DC.

The final drawback is one we've already discussed, and that is situations where we want to make multiple hopes. Quite frankly, NTLM leaves us with no options. We have to make each hop different from the previous one, whether we like it or not. Kerberos delegation ensures we can pass the credentials through all the hops until we reach the final destination.

What Is an SPN, Why Do I Need to Configure It, and How Do I Do So?

A Service Principal Name (SPN) provides the information to the client about the service. Basically, each SPN consists of 3 or 4 pieces of information:

- The type of service (for SQL Server it is called MSSQLSvc)
- The name of the server
- The port (if this needs to be specified)
- The service account running the service.

All of these need to match up for the client to be able to validate the service. If any of these are wrong, Kerberos authentication won't happen. In some cases, we'll get that SSPI context error and in fact, SSPI won't even drop back to using NTLM, meaning we don't connect at all. Therefore, the key is to get everything correct when we set the SPN.

In order to set an SPN, you must either be a Domain Admin level user or you must be the computer System account (or an account that talks on the network as the System account, such as the Network Service account). Typically, we advise that SQL Server should be run as a local or domain user, so that rules out the second case. We also advise that SQL Server shouldn't be a domain admin level account, and that rules out the first case. What this means is a domain admin level account will need to set the SPN manually. Thankfully, Microsoft provides a nice utility called SETSPN in the Support Tools on the OS CD/DVD to do so. It can also be downloaded from the Microsoft site.

Using SETSPN

SETSPN has three flags we're interested in:

- **-L** : This lists the SPNs for a particular account

- **-A** : This adds a new SPN
- **-D** : This deletes an existing SPN

The key to understanding SPNs is to realize they are tied to an account, whether that be a user or computer account. If we want to see what SPNs are listed for a particular account, here is the syntax:

```
SETSPN -L <Account>
```

For instance, if I have a server called MyWebServer, I can list the SPNs assigned to that computer account by:

```
SETSPN -L MyWebServer
```

If, instead, I am running my SQL Server under the MyDomain\MyServiceAccount user account, I can check the SPNs listed for that account by:

```
SETSPN -L MyDomain\MyServiceAccount
```

To add an SPN, it's important that we know the service account SQL Server is running under. Also, it is important to know the TCP port SQL Server is listening on. If it's a default instance, the port by default is 1433, although this can be changed. If it's a named instance, unless we have gone in and manually set a static port, SQL Server could change the port at any time. Therefore, it's important to set a port statically. I've described how to do so in the a blog post. Once we have those bits of information, we can add an SPN via the following syntax:

```
SETSPN -A MSSQLSvc/<SQL Server Name>:<port> <account>
```

If we're dealing with a default instance listening on port 1433, we can leave off the :<port> (but it is still a good idea to have an entry both with and without the port). One other thing to remember is it is important to specify SPNs for both the NetBIOS name (e.g. MySQLServer) as well as the fully qualified domain name (e.g. MySQLServer.mydomain.com). So applying this to a default instance on MyDBServer.mydomain.com running under the service account MyDomain\SQLServerService, we'd execute the following commands:

```
SETSPN -A MSSQLSvc/MyDBServer MyDomain\SQLServerService
SETSPN -A MSSQLSvc/MyDBServer:1433 MyDomain\SQLServerService
SETSPN -A MSSQLSvc/MyDBServer.mydomain.com MyDomain\SQLServerService
SETSPN -A MSSQLSvc/MyDBServer.mydomain.com:1433 MyDomain\SQLServerService
```

For a named instance, we typically only require two commands, because there isn't a case where a client is just connecting to the name of the server. For instance, let's assume we have a named instance called Instance2 listening on port 4444 on that same server using that same service account. In that case we'd execute the following commands:

```
SETSPN -A MSSQLSvc/MyDBServer:4444 MyDomain\SQLServerService
SETSPN -A MSSQLSvc/MyDBServer.mydomain.com:4444 MyDomain\SQLServerService
```

And in those rare cases where we need to delete an SPN (for instance, we change the service account or switch ports), we can use the -D switch. It's syntax is parallel to the -A switch:

```
SETSPN -D MSSQLSvc/<SQL Server Name>:<port> <account>
```

I've Done All of That. How Can I Verify Logins Are Connecting Via Kerberos?

Within SQL Server there is a very simple query we can execute to determine what type of authentication was performed on each connection. Here's the query:

```
SELECT
    s.session_id
  , c.connect_time
  , s.login_time
  , s.login_name
  , c.protocol_type
  , c.auth_scheme
  , s.HOST_NAME
  , s.program_name
FROM sys.dm_exec_sessions s
  JOIN sys.dm_exec_connections c
    ON s.session_id = c.session_id
```

The query returns a lot of information to help you identify the connections. The connect_time and login_time should be pretty close together and it gives you a window of when the initial connection was made. The login_name, along with host_name and program_name, help you identify the exact login. From there the protocol_type helps you narrow down the connection if you have different endpoints for your SQL Server other than just TSQL (for instance, mirroring or HTTP). And finally, the auth_scheme will reveal, for a Windows account, what security protocol was used. If Kerberos authentication was successful, you should see the auth_scheme reflect Kerberos instead of NTLM.