Thrift and Security

**Security issues**

Security is an important issue when dealing with services that are exposed to clients. It doesn't matter if a local service is available only to clients in the local network, or if it is a public API used by millions; security is always something that shouldn't be ignored. Even when designing internal services, you shouldn't assume that the environment is secure by definition. Other machines in the network may be compromised or behave erratically.

**General security tips**

There are some general security tips that apply to every service, regardless of whether it is based on Apache Thrift or some other technology or framework. You should remember them even if you develop a simple website.

The two most important elements are **authentication** and **authorization**. Though they are often confused with each other, they don't mean the same thing. Authentication is the process whereby the identity of a connecting client (be it a person, a service, an application, and so on) is confirmed. In the simplest and most common scenario, it is a pair of usernames and passwords. These parameters are checked against records in the database and, if they match, the client is considered authenticated. Quite often, services use the API key instead of the username/password pair. Such a key is a long, alphanumeric string (for example, the Amazon Web Service API key may look like wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY). This solution is considered to be more secure as these keys are generally longer than a regular password; therefore, they are harder to break, may be freely assigned and revoked (it is possible to have multiple keys per account), or can even have different permissions in the scope of a single user account. Due to this, you may consider using API keys instead of the username/ password pair when authenticating clients.

Authorization is a process of defining an access policy. So, when we have an already authenticated client (this means that we know its identity), we need to assess what the client is permitted to do. As with any other system, not everyone is allowed to perform any action. Based on internal records, you should grant the client access to only those actions that are allowed to be performed.

Especially when exposing services externally, you should carefully plan which actions should be available for anonymous users. This is strongly dependent on business needs, but as a rule, this practice should be avoided, for example, exposing large sets of data in bulk for download. Let's imagine that you have a classified ad service; you don't want someone to download your whole ad set and publish it on their website, right?

Data privacy is another important security rule for any application in general, but especially the service. This means that users should have access only to data that they are supposed to see. For example, a customer should have access only to his transactions; he shouldn't be able to view other peoples' transactions, for example, by substituting the transaction identifier variable.

**Transport Layer Security/Secure Sockets Layer**

We earlier discussed securing the service when the client has access to it, but what about the connection itself? In many environments, communication between the client and the server may be eavesdropped on by an attacker. To provide privacy to the transmission, we need to use the **Transport Layer Security** (**TLS**) protocol (frequently referred to as **SSL** or **TLS/SSL**). This protocol is commonly used to secure web applications, e-mail communication, messaging, and so on. You may know it from your experience, for example, when visiting websites whose address begins with https://.

Without going into much detail, security when using TLS is derived from the fact that the transmission is encrypted.

We will be working on the Java example code that we have already used in the previous section on multiplexing.

You can view the code from the code folder. The files are named MySecureMultiplexedServer.java and MySecureMultiplexedClient.java.

**Generating keystores**

To have communication encrypted, we need to have a keystore on the server (containing the public and private keys and the certificate) and a truststore with the certificate and the public key on the client.

We will use the keytool (shipped with Java) to create both of these files. Our certificates will be self-signed.

First, let's create a keystore. To do this, use the following command. You will be asked to enter a new password for the keystore and to provide some information about the unit issuing the key. This information is included in the certificate, so it should be valid. At the end, you need to type yes to confirm the validity of the information:

**$ keytool -genkeypair -alias mykey -keyalg RSA -validity 365**

**Enter keystore password:**

**Re-enter new password:**

**What is your first and last name?**

**[Unknown]: John Smith**

**What is the name of your organizational unit?**

**[Unknown]: IT Department**

**What is the name of your organization?**

**[Unknown]: ACME Corporation**

**What is the name of your City or Locality?**

**[Unknown]: Warsaw**

**What is the name of your State or Province?**

**[Unknown]: mazowieckie**

**What is the two-letter country code for this unit?**

**[Unknown]: PL**

**Is CN=John Smith, OU=IT Department, O=ACME Corporation, L=Warsaw, ST=mazowieckie, C=PL correct?**

**[no]: yes**

**Enter key password for <mykey>**

**(RETURN if same as keystore password):**

The parameters of the keytool -genkeypair command are:

1. -alias mykey: This is the name of the keystore; it can be anything you wish
2. -keyalg RSA: This is the key that will be generated using the RSA algorithm
3. -validity 365: This key will be valid for 365 days (after this, you need to generate a new key)

The key is saved by default in the .keystore file in the directory in which you ran this command.

Now, we need to export the certificate from the keystore. We will need the certificate to create the truststore. To export the certificate, run the following command:

**$ keytool -export -alias mykey -keystore .keystore -rfc -file certificate.cer**

**Enter keystore password:**

**Certificate stored in file <certificate.cer>**

You were asked about the password to the keystore (the one that you noted down in the previous step), and the certificate was stored in the certificate.cer file.

Now, here's the last step: let's create the truststore. Run the following command. You will be asked for the keystore's password and to confirm (by writing yes) that you want to trust the given certificate:

**$ keytool -import -alias mykey -file certificate.cer -keystore .truststore**

**Enter keystore password:**

**Re-enter new password:**

**Owner: CN=John Smith, OU=IT Department, O=ACME Corporation, L=Warsaw, ST=mazowieckie, C=PL**

**Issuer: CN=John Smith, OU=IT Department, O=ACME Corporation, L=Warsaw, ST=mazowieckie, C=PL**

**Serial number: 59a9be54**

**Valid from: Tue Nov 17 21:40:49 CET 2015 until: Wed Nov 16 21:40:49 CET 2016 Certificate fingerprints:**

**MD5: 6F:58:7B:89:13:BB:52:75:33:C6:09:78:91:CD:33:89**

**SHA1: F9:DE:D5:BB:29:50:9E:8F:05:20:C6:7F:9D:F5:13:5F:2D:EA:61:00**

**SHA256: 55:2E:2A:31:07:08:06:23:F8:42:43:3F:C0:E7:FB:6C:07:38:CD:AB:02 :5C:28:BC:49:87:E3:6E:2B:38:05:AD**

**Signature algorithm name: SHA256withRSA**

**Version: 3**

**Extensions:**

**#1: ObjectId: 2.5.29.14 Criticality=false**

**SubjectKeyIdentifier [**

**KeyIdentifier [**

**0000: 18 78 04 47 98 6B 68 4F 22 33 E4 F0 C0 AF CF B5 .x.G.khO"3......**

**0010: 76 0A 01 82 v...**

**]**

**]**

**Trust this certificate? [no]: yes**

**Certificate was added to keystore**

If you did everything properly, the information would be saved to the .truststore file.

At any time, you can check your keystores using the following command; after you provide the store's password, its contents will be listed:

**$ keytool -list -keystore .truststore**

**Enter keystore password**

**Keystore type: JKS**

**Keystore provider: SUN**

**Your keystore contains 1 entry**

**mykey, 17-Nov-2015, trustedCertEntry,**

**Certificate fingerprint (SHA1): F9:DE:D5:BB:29:50:9E:8F:05:20:C6:7F:9D:F5 :13:5F:2D:EA:61:00**

Now, let's use these keystores in our code.

**Using keystores in the Java code**

Let's begin with the server's code. Get the code from the MyMultiplexedServer. java file and copy it to MySecureMultiplexedServer.java; remember to also update the class name to MySecureMultiplexedServer, and add following imports:

import org.apache.thrift.transport.TSSLTransportFactory;

import org.apache.thrift.transport.TSSLTransportFactory. TSSLTransportParameters;

We will be substituting the myserver method. Instead of its original content, let's use the following code:

public static void myserver(TMultiplexedProcessor processor) {

// create parameters store for TSSLTransport

TSSLTransportParameters params = new TSSLTransportParameters();

// point to the keystore, provide keystore's password

// remember about giving the proper path

params.setKeyStore(".keystore", "somepassword", null, null);

// construct the transport, server and start serving

TServerTransport serverTransport = TSSLTransportFactory. getServerSocket(8081, 0, null, params);

TServer server = new TThreadPoolServer(new TThreadPoolServer. Args(serverTransport).processor(processor));

System.out.println("Starting secure multiplexed server on port 8081...");

server.serve();

}

To serve our service over a secure connection, we need to provide the path to our keystore file and the password. It will be run on port 8081 of the localhost.

Now, let's get to the client; you need to perform very similar work to the one you did for the server. First, get the code from the MyMultiplexedClient.java file and copy it to MySecureMultiplexedClient.java; remember to also update the class name to MySecureMultiplexedClient. Then, add the imports:

import org.apache.thrift.transport.TSSLTransportFactory;

import org.apache.thrift.transport.TSSLTransportFactory. TSSLTransportParameters;

We will be substituting the following—not secure—part of the code:

TTransport transport = new TSocket("localhost", 8080);

transport.open();

Instead, we will use the code that will let us connect to the secure server running on port 8081:

TSSLTransportParameters params = new TSSLTransportParameters();

// point to the keystore, provide keystore's password

// remember about giving the proper path

params.setTrustStore(".truststore", "somepassword");

// construct the transport

transport = TSSLTransportFactory.getClientSocket("localhost", 8081, 0, params);

Note that in this case, there's no need to use transport.open() as this function is called by the transport method.

Now you have everything that you need to run the secure server and client.