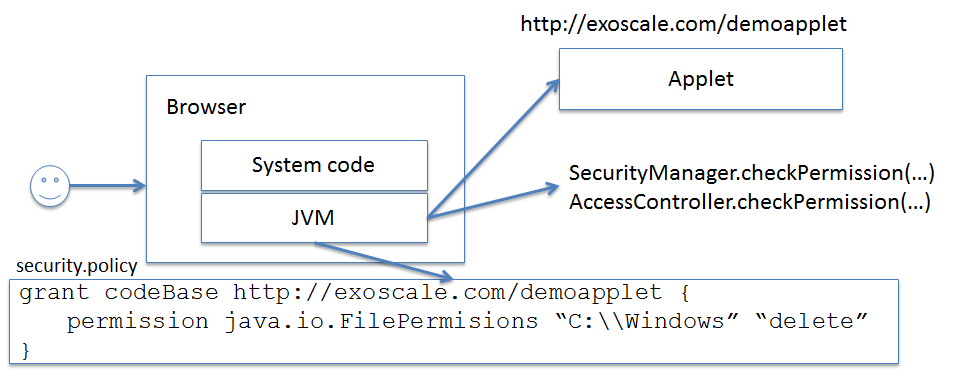
State of security in JDK 11

The Java security sandbox model has been around since more than a decade and with some enhancements over the years it remains pretty much the same as of JDK 11. It dates back to JDK 1.2 and is described succinctly by the following diagram:



In a nutshell the security sandbox model was introduced out of the necessarity to provide a constraint environment for source code loaded from java applets running on the customer's environment (the browser). Later on this model expanded to pretty much any Java managed environment capable to load and execute custom application (such as OSGi, Java EE or even Java stored procedures running in Oracle RSBMS).

Every time when a permission check needs to be done by the application either an instance of the **java.lang.SecurityManager** class installed in the JVM runtime or the static methods ot the **java.security.AccessController** utility need to be used. In fact the **java.lang.SecurityManager** delegates its operations to the **java.security.AccessController**. To describe the model briefly consider the following flow:

* the Java application is being loaded by the JVM and during classloading for each class of the application a **java.security.ProtectionDomain** instance is set that contains the code source (location of where the class was loaded from) and set of Java permissions that apply to that class based on its location (as specified in the **security.policy** file);
* during execution the **checkPermission** method of the installed security manager (or AccessController utility) is called to check if the calling code has the permission to perform the requested action identified by a particular Java permission (instance of the **java.security.Permission**) class. The permission check is done by traversing the current call stack and checking whether the protection domain of each class in the call stack has the requested permission (passed as a parameter to the **checkPermission** call). If there is even a single class that does have the permissions there is a security exception raised;
* on certain occasions there is a need to escalate privileges of a block of code (typically to allow temporary access to an operation that is forbidden to callers by default). That can be achieved by calling one of the **doPrivileged** methods

Apart from the security sandbox model the JDK is bringing a number of security utilities for use by developers such as:

* JCA (Java Cryptography Architecture). Provides a number of utilities for performing cryptographic operations such as creating digital signatures and message digests, using symmetric/asymmetric block/stream cyphers and other types of cryptographic services and algorithms. The JCA is provider-based, not bound to a particular set of algorithm implementations. There is a default implementation of the provider API provided by the JDK called SunJCA. Another widely used implementation is provided by the BouncyCastle library.
* PKI (Public Key Infrastructure) utilities. These are utilities for working with digital certificates, certificate revocation lists (CRLs) or the OCSP (Online Certificate Status Protocol). Later two provide mechanisms for checking of certificate revocation status as provided by the certificate authority. In that category are also the operations used to deal with key and trust stores in different formats (such as JKS and PKCS12);
* JSSE (Java Secure Socket Extension). This is JDK's implementation of the TLS (formerly SSL) and as of recently also DTLS series of protocols. In fact the JSSE is the one with the most improvements as of latest versions of the JDK as we shall see in the next sections;
* Java GSS API (Java Generic Security Services). The GSS API can be considering an alternative to the JSSE API and it uses token-based communication for encryption of traffic;
* Java SASL API (Java Simple Authentication and Security Layer). The framework provides a generic mechanism to establish the authentication channel between a client and a server.

# All of these have been around for quite some time and enhanced throughout the JDK versions. Developers still tend to choose in addition a number of third-party security libraries and frameworks that typically provide extra security capabilities missing from the JDK. For example the BouncyCastle library provides enhanced cryptography algorithms and utilities and the JSch library provides SSH support for application (which is missing from the JDK).

# We will further look into the state of JDK security as of JDK 11 and detail what are the benefits for developers.

# JDK 9, 10 and 11 from a security perspective

One of the first things that one can think of in terms of security when we deal with distributed systems is the necessity to establish secure communication channel between the distinct components of that system. The de-facto standard that has been well established over the years for the purpose is the transport layer security series of protocols (TLS for short) providing a number of improvements over its predecessor SSL such as:

* SSL protocols have exposed vulnerabilities such as the notorious POODLE attack over SSL 3.0 which is also made possible in modern browsers due to automatic downgrade to SSL 3.0 (if not disabled) used for the purpose of interoperability. Other vulnerabilities such as BEAST even cross the boundary by exploiting not only SSL (3.0) but also TLS 1.0 (but not later versions).

* TLS certificates are cryptographically stronger and hence more difficult to exploit than the SSL certificates;

Without diving into more details it is worth saying that the changes made in regard to TLS break interoperability with the earlier SSL series of the protocol. Major enhancements have been introduced in the JSSE API in regard to the TLS support in the JDK. Let’s see how they fill in some of the gaps in the protocol’s capabilities:

* The TLS protocol is working over TCP meaning that it provides out of the box reliability of transfer (with retransmission of failed packets), error detection, flow and congestion control. What about UDP protocols such as SIP (used by messaging applications) or DNS (used for name resolution)? DTLS comes to the rescue: the introduction of a **datagram transport layer security protocol** in the JDK enables applications to establish secure communication over an unreliable protocol such as UDP;
* TLS is typically bound to a concrete application protocol: for HTTP we have HTTPS, for the FTP we have SFTP and son on. These require distinct ports for each distinct protocol running over TLS. What about different versions of the same protocol? Same problem. ALPN comes to the rescue – the **application layer protocol negotiation** extension of TLS enables applications to negotiate the application protocol for use by communicating parties during the TLS handshake that establishes the secure channel. Consider for example HTTP 1.1 and HTTP 2.0: a TLS client and server may negotiate which version of the HTTP protocol to use during the TLS handshake process;
* When it comes to secure communication we cannot ignore thinking about performance implications as well. Although latest hardware enhancements minimize the impact on applying TLS over an existing protocol there are still some areas such as certificate revocation checking that imply increased network latency and hence are a performance hit. OCSP stapling is an improvement of the TLS protocol that allows for certificate revocation checking requests from the TLS client to the certificate authority that use the OCSP protocol to be performed by the TLS server thus minimizing the number of requests to the certificate authority.

All of the above are introduced in JDK 9 while in JDK 10 a default set of root certificate authorities have been provided to the trust store of the JDK for use by Java applications. Moving further indisputably the security highlight of JDK 11 is the implementation of major parts of the TLS 1.3 protocol in JSSE. Major benefits of TLS 1.3 are improved security and performance including the following enhancements introduced by the protocol (as highlighted by [JEP 332](https://openjdk.java.net/jeps/332) upon which the implementation is based):

* Enhanced protocol version negotiation using an extension field indicating the list of supported versions;
* Improved full TLS handshake for both client and server sides (more compact than in TLS 1.3);
* Improved session resumption using a PSK (Pre-Shared Key) extension;
* The possibility to update cryptographic keys and corresponding initialization vectors (IV) after a Finish request is send in the TLS handshake;
* Additional extensions and algorithms;
* Two new cipher suites: TLS\_AES\_128\_GCM\_SHA256 and TLS\_AES\_256\_GCM\_SHA384
* RSASSA-PSS signature algorithms;

It is good to note that TLS 1.3 is not backward compatible with previous versions of the protocol but the JSSE implementation provides backward compatibility mode. In addition both the synchronous (via the SSLSocket API) and asynchronous mode of operation (via the SSLEngine API) are updated to support TLS 1.3.

We want to build a client-server application that is using a custom application protocol secured with the latest version of TLS. We can use JDK 11 and provide an implementation based on JSSE (already providing support for TLS1.3). The following example TLS 1.3 client and server may be a good starting point:

TLSv1.3 JSSE server

System.setProperty("javax.net.ssl.keyStore", "C: /sample.pfx");

System.setProperty("javax.net.ssl.keyStorePassword", "sample");

SSLServerSocketFactory ssf = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

SSLServerSocket ss = (SSLServerSocket) ssf.createServerSocket(4444);

ss.setEnabledProtocols(new String[] {"TLSv1.3"});

ss.setEnabledCipherSuites(new String[] {"TLS\_AES\_128\_GCM\_SHA256"});

while (true) {

SSLSocket s = (SSLSocket) ss.accept();

SSLParameters params = s.getSSLParameters();

s.setSSLParameters(params);

BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));

String line = null;

PrintStream out = new PrintStream(s.getOutputStream());

while (((line = in.readLine()) != null)) {

System.out.println(line);

out.println("Hi, client");

}

in.close();

out.close();

s.close();

}

TLSv1.3 JSSE client

try {

System.setProperty("javax.net.ssl.trustStore", "C:/sample.pfx");

System.setProperty("javax.net.ssl.trustStorePassword", "sample");

SSLSocketFactory ssf = (SSLSocketFactory) SSLSocketFactory.getDefault();

SSLSocket s = (SSLSocket) ssf.createSocket("127.0.0.1", 4444);

s.setEnabledProtocols(new String[] {"TLSv1.3"});

s.setEnabledCipherSuites(new String[] {"TLS\_AES\_128\_GCM\_SHA256"});

SSLParameters params = s.getSSLParameters();

s.setSSLParameters(params);

PrintWriter out = new PrintWriter(s.getOutputStream(), true);

out.println("Hi, server.");

BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));

String x = in.readLine();

System.out.println(x);

out.close();

in.close();

s.close();

} catch (Exception ex) {

ex.printStackTrace();

}

As you can see from the highlighted lines above in order to enable TLS 1.3 in JDK 11 we need to specify the appropriate constant representing the protocol version over the server and client SSL sockets created and also set the appropriate cypher suites that TLS 1.3 expects again on both the client and server SSL sockets.

# Enhancing the Security Sandbox Model

Now let’s assume that our TLS 1.3 server is implemented as a set of distinct Jigsaw modules (Jigsaw brings a new module system to the JDK ecosystem as of JDK 9). Moreover our server application provides a plug-in system that uses a distinct classloader to load different applications providing integrations with third-party systems. This would imply that we need a form of access control for the applications managed by our server application. The Java security sandbox comes to the rescue. It remains the same as of JDK 9 where Jigsaw modules bring subtle changes to the JDK so that the permission model can be applied to modules. This is achieved by simply introducing a new scheme (**jrt**) for referring to Jigsaw modules when specifying permissions in the security.policy file. For example if we install a security manager in the JSSE server we created by adding the following at the beginning of the source code:

System.setSecurityManager(new SecurityManager());

When you rerun the JSSE server you will get:

java.security.AccessControlException: access denied ("java.util.PropertyPermission" "javax.net.ssl.trustStore" "write")

This is simply because we also need to put the proper permissions in the security.policy file residing in the JDK installation directory (by default that is under conf/security) for the JDK we use to run the JSSE server. If the codebase (location from where we start the JSSE server) is the compilation directory (i.e. we run the JSSE server from the compiled Java class containing the snippet) in the security.policy file we would end up with (adding a few more permissions required):

grant codeBase "file:/C:/project/target/" {

permission java.util.PropertyPermission "javax.net.ssl.keyStore", "write";

permission java.util.PropertyPermission "javax.net.ssl.keyStorePassword", "write";

permission java.net.SocketPermission "localhost:4444", "listen,resolve";

};

If our JSSE server was packaged as a JDK module named “com.exoscale.jsse.server” we could have specified the above entry in the following format:

grant codeBase "jrt:/com.exoscale.jsse.server " {

…

};

# Deploying the sample application

We are going to demonstrate is manual deploy which of course can be automated with proper tools. Assuming you have provisioned an Linux Ubuntu 18.04 LTS 64-bit (i.e. from the Exoscale Web UI) you can ssh to the machine and do the following (assuming we have bundled our application in a runnable JAR and having it uploaded it somewhere accessible from your VM):

sudo apt-get update  
sudo add-apt-repository ppa:linuxuprising/java  
sudo apt-get install oracle-java11-installer  
wget <https://filebin.net/vxsv072m3jebv90m/jsseserver.jar?t=ba3n293q> –O jsseserver.jar  
java –jar jsseserver.jar

Of course you need to also specify the proper permissions in the security.policy file of the installed JDK in case you run your JSSE server with a security manager installed.

# Conclusion

We saw briefly how to get started with TLS 1.3 in JDK 11, how to further apply the JDK security sandbox model over your application and how to provision your JDK application on the Exoscale cloud. Further JDK versions will continue providing security improvements.