Platform SDK Developer's Guide

TLS and the Platform SDK Commons Library

5/13/2025

TLS and the Platform SDK Commons Library

Platform SDK for Java

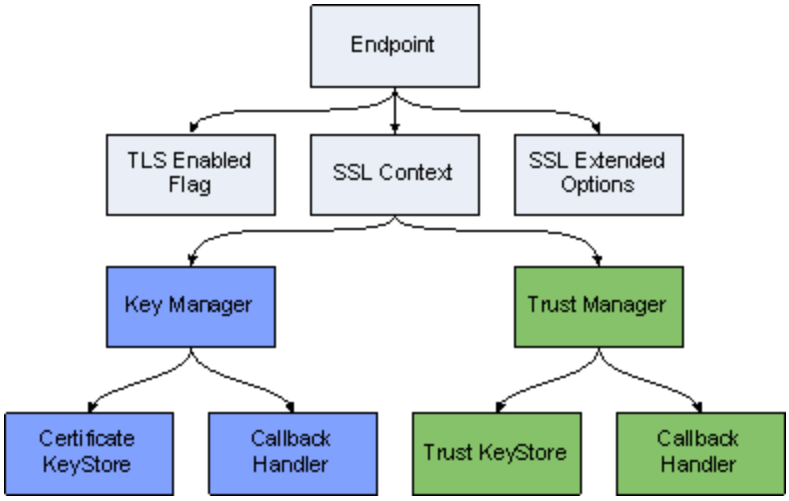
> **Important**
> The contents of this page only apply to Java implementations.

Using the Platform SDK Commons Library to Configure TLS

Starting with Platform SDK 8.1.1, the only way to configure connections is by using `Endpoint` objects, which contain all parameters related to the endpoint connection—including TLS parameters that indicate whether TLS is enabled and provide details about the SSL context and extended options.

> **Tip**
> In earlier releases, Platform SDK provided three ways to configure connections:
>
> - using `ConnectionConfiguration` objects passed to `Protocol` constructors
> - setting parameters in the protocol context
> - adding a textual parameter representation to the URL query

The following diagrams show interdependencies among the Platform SDK objects used to establish network connections and support TLS.



TLS Configuration Objects Containment Hierarchy

This page outlines each step required to create supporting objects for a TLS-enabled `Endpoint`.

Callback Handlers

In many cases, certificate or key storage is password-protected. This means that Platform SDK will need the password to access storage. The Java `CallbackHandler` interface offers a flexible way to pass this type of credential data:

```
package javax.security.auth.callback;
...
public interface CallbackHandler {
    void handle(Callback[] callbacks)
        throws java.io.IOException, UnsupportedCallbackException;
}
```

The `handle()` method accepts credential requests in the form of `Callback` objects that have appropriate setter methods. The most common callback implementation is `PasswordCallback`. User code may use a GUI to ask the end user to:

- enter a password
- retrieve a password from a file, pipe, network, and so on

Here is an example of a `CallbackHandler` delegating password retrieval to a GUI:

```
CallbackHandler callbackHandler = new CallbackHandler() {
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (Callback c : callbacks) {
            if (c instanceof PasswordCallback) {
                PasswordCallback p = (PasswordCallback) c;
                p.setPassword(gui.getKeyStorePassword());
            }
        }
    }
};
```

When No Password is Required

In some cases, certificate storage does not need a password. The API may still dictate that a `CallbackHandler` be provided however, so the Platform SDK includes a predefined class that can be used as a "dummy" `CallbackHandler` for this scenario:

```
com.genesyslab.platform.commons.connection.tls.DummyPasswordCallbackHandler
```

Here is an example of using this dummy class:

```
CallbackHandler callbackHandler = new DummyPasswordCallbackHandler();
```

Key Managers

Java provides a `KeyManager` interface. This interface defines functionality that can be used to load and contain certificates or keys, or to select appropriate certificates or keys.

Classes based on the `KeyManager` interface are used by Java TLS support to retrieve certificates that will be sent over the network to a remote party for validation. They are also used to retrieve the corresponding private keys. On the client side, `KeyManager` classes retrieve client certificates or keys; on the server side they retrieve server certificates or keys.

The Platform SDK Commons library has a helper class, `KeyManagerHelper`, which makes it easy to create key managers using several types of key stores and security providers. The built-in key manager types are:

- **PEM** — reads certificate/key pairs from X.509 PEM files.
- **MSCAPI** — uses the Microsoft CryptoAPI and Windows certificate services to retrieve certificate/key pairs.
- **PKCS11** — delegates to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — retrieves a certificate/key pair from a Java Keystore file.
- **Empty** — does not retrieve anything. This type is for use as a dummy key manager. For example, clients that do not have certificates can use it.

Here are some examples of key manager creation:

```
// From PEM file
X509ExtendedKeyManager km = KeyManagerHelper.createPEMKeyManager(
        "c:/cert/client-cert.pem", "c:/cert/client-cert-key.pem");

// From MSCAPI
CallbackHandler cbh = new DummyPasswordCallbackHandler();
// Whitespace characters are allowed anywhere inside the string
String certThumbprint =
        "4A 3F E5 08 48 3A 00 71 8E E6 C1 34 56 A4 48 34 55 49 D9 0E";
X509ExtendedKeyManager km = KeyManagerHelper.createMSCAPIKeyManager(
        cbh, certThumbprint);

// From PKCS11
// This provider does not allow customization of Key Manager
// This is required for FIPS-140 certification
// Dummy callback handler will not work, must use strong password
CallbackHandler passCallback = ...;
```

```
X509ExtendedKeyManager km = KeyManagerHelper.createPKCS11KeyManager(
        passCallback);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Individual entries in JKS key store can be password-protected
char[] keyStorePass = "keyStorePass".toCharArray();
char[] entryPass = "entryPass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSKeyManager(
        "c:/cert/client-cert.jks", keyStorePass, entryPass);

// Empty key manager
// Using KeyManagerHelper class
X509ExtendedKeyManager km1 = KeyManagerHelper.createEmptyKeyManager();
// Direct creation
X509ExtendedKeyManager km2 = new EmptyX509ExtendedKeyManager();
```

Trust Managers

A Trust Manager is an entity that decides which certificates from a remote party are to be trusted. It performs certificate validation, checks the expiration date, matches the host name, checks the certificate against a CRL list, and builds and validates the chain of trust. The chain of trust starts from a certificate trusted by both sides (for example, a CA certificate) and continues with second-level certificates signed by CA, then possibly with third-level certificates signed by second-level authorities and so on. Chain length can vary, but Platform SDK was designed to explicitly support two-level chains consisting of a CA certificate and a leaf certificate signed by CA.

Trust manager instances are created based on storage that contains trusted certificates. The number of trusted certificates can vary depending on the type of trust manager being used. With PEM files, the storage contains only a single CA certificate; other provider types can have larger sets of trusted certificates.

The Platform SDK Commons library has a helper class, `TrustManagerHelper`, which makes it easy to create trust managers that use several types of certificate stores and security providers, and which can accept additional parameters that affect certificate validation. Built-in trust manager types are:

- **PEM** — Reads a CA certificate from an X.509 PEM file.
- **MSCAPI** — Uses the Microsoft CryptoAPI and Windows certificate services to retrieve CA certificates and validate certificates.
- **PKCS11** — Delegates certificate validation to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — Retrieves a CA certificate from a Java Keystore file and uses Java built-in validation logic.
- **Default** — Uses trusted certificates shipped with or configured in Java Runtime and Java built-in validation logic.
- **TrustEveryone** — Trusts any certificates. Can be used on the server side when you do not expect any certificates from clients, or during testing.

Here are some examples of trust manager creation (with generic `crlPath` and `expectedHostName` parameters defined in the first example):

```
// Generic parameters for trust manager examples
String crlPath = "c:/cert/ca-crl.pem";
String expectedHostName = "serverhost";
// From PEM file
```

```
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
        "c:/cert/ca.pem", crlPath, expectedHostName);

// From MSCAPI
// CRL is loaded from PEM file (Platform SDK supports only file-base CRLs)
// Concrete CA is not specified, all certificates from WCS Trusted Root are used
CallbackHandler cbh = new DummyPasswordCallbackHandler();
X509TrustManager tm = TrustManagerHelper.createMSCAPITrustManager(
        cbh, crlPath, expectedHostName);

// From PKCS#11
// This provider implementation in Java does not allow custom host name check,
// but CRL can still be used
X509TrustManager tm = TrustManagerHelper.createPKCS11TrustManager(
        cbh, crlPath);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Certificate-only entries cannot have passwords in JKS key store
// CRL and host name check are supported
char[] keyStorePass = "keyStorePass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSTrustManager(
        "c:/cert/ca-cert.jks", keyStorePass, crlPath, expectedHostName);

// From Java built-in trusted certificates
// This one does not support CRL and host name check
X509ExtendedKeyManager km = KeyManagerHelper.createDefaultTrustManager();

// Trust Everyone
X509ExtendedKeyManager km =
        KeyManagerHelper.createTrustEveryoneTrustManager();
```

SSLContext and SSLExtendedOptions

An `SSLContext` instance serves as a container for all SSL and TLS parameters and objects and also as a factory for `SSLEngine` instances.

`SSLEngine` instances contain logic that deals directly with TLS handshaking, negotiation, and data encryption and decryption. `SSLEngine` instances are not reusable and must be created anew for each connection. This is a good reason for requiring users to provide an `SSLContext` instance rather than an instance of `SSLEngine`. `SSLEngine` instances are created by the Platform SDK connection layer and are not exposed to user code.

Only some of the parameters for `SSLEngine` can be pre-set in `SSLContext`. However, the `SSLExtendedOptions` class may be used to collect additional parameters.

`SSLExtendedOptions` currently contains two parameters:

- the "mutual TLS" flag
- a list of enabled cipher suites

The mutual TLS flag is used only by server applications. When the flag is turned on, the server will require connecting clients to send their certificates for validation. The connections of any clients that do not send certificates will fail.

The list of enabled cipher suites contains the names of all cipher suites that will be used as filters for `SSLEngine`. As a result, only ciphers that are supported by `SSLEngine` and that are contained in the

enabled cipher suites list will be enabled for use.

Platform SDK includes the `SSLContextHelper` helper class to support one-line creation of `SSLContext` and `SSLExtendedOptions` instances.

Here are some examples:

```
// Creating SSLContext
KeyManager km = ...;
TrustManager tm = ...;
SSLContext sslContext = SSLContextHelper.createSSLContext(km, tm);

String[] cipherList = new String[] {
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
        "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA"};
// Can be single String with space-separated suite names
String cipherNames = "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA " +
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA " +
        "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA";
boolean mutualTLS = false;

// Creating SSLExtendedOptions directly
SSLExtendedOptions sslOpts1 =
        new SSLExtendedOptions(mutualTLS, cipherList);
SSLExtendedOptions sslOpts2 =
        new SSLExtendedOptions(mutualTLS, cipherNames);

// Create SSLExtendedOptions using the helper class:
SSLExtendedOptions sslOpts3 =
        SSLContextHelper.createSSLExtendedOptions(mutualTLS, cipherList);
SSLExtendedOptions sslOpts4 =
        SSLContextHelper.createSSLExtendedOptions(mutualTLS, cipherNames);
```

Endpoints

Now that supporting objects have been created and configured, you are ready to create an `Endpoint`.

The connection configuration parameters of an `Endpoint` are read-only—they cannot be changed after the `Endpoint` is created. This configuration information is then used by `Protocol` instances, the warm standby service, the connection layer and the TLS layer.

A sample `Endpoint` configuration is shown below:

```
ConnectionConfiguration connConf = ...;
SSLContext sslContext = ...;
SSLExtendedOptions sslOpts = ...;
tlsEnabled = true;
// Specifying host name and port.
Endpoint ep1 = new Endpoint("Server-1", "serverhost", 9090, connConf,
        tlsEnabled, sslContext, sslOpts);
// Specifying URI. Query part is still supported.
String uri = "tcp://Server-1@serverhost:9090/" +
        "?protocol=addp&addp-remote-timeout=5&addp-trace=remote";
Endpoint ep2 = new Endpoint("Server-1", uri, connConf,
        tlsEnabled, sslContext, sslOpts);
```

Note: Configuration parameters can be set directly in a `Protocol` instance context, but will be overwritten and lost under the following conditions:

- a new Endpoint is set up
- the protocol is forced to reconnect
- a warm standby switchover occurs

Configuring TLS for Client Connections

Using the information above, you are now ready to configure actual client connections.

Example:

```
// Get TLS configuration objects for connection
String clientName = "ClientApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSLExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
        clientName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

TServerProtocol tsProtocol = new TServerProtocol(epTSrv);
tsProtocol.setClientName(clientName);
tsProtocol.open();
```

Configuring TLS for Servers

Using the information above, you are now ready to configure actual server connections.

```
String serverName = "ServerApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSLExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
        serverName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

ExternalServiceProtocolListener serverChannel =
        new ExternalServiceProtocolListener(endpoint);
```

Parameter-based TLS Configuration

Platform SDK has a way to create TLS objects based on a set of parameters in a more declarative fashion rather than creating them programmatically. This feature was initially developed as a part of Application Template to configure TLS based on parameters from Configuration objects and then was generalized to use different parameter sources and moved to Commons. Currently this mechanism supports only three providers: PEM, MSCAPI and PKCS#11. Usage sequence is the following:

1. Prepare a source of TLS parameters and parse it using TLSConfigurationParser resulting in TLSConfiguration instance.

2. Customize TLSConfiguration.
    1. Add callback handlers.
    2. Clients: set expected host name.
3. Create SSLContext and SSLExtendedOptions from TLSConfiguration.

This section continues with step-by-step examples and ends with a more detailed review of helper classes.

Parsing TLS Parameters

Platform SDK Commons has a few helper classes that make it easier to extract TLS parameters from a properties files, command-line arguments, etc.: TLSConfiguration and TLSConfigurationParser. TLSConfiguration is a container for parsed TLS parameters and TLSConfigurationParser provides a general parsing method and several overloaded shortcut methods for specific cases.

Examples:

```
// Using KVList as a parameters source
KVList tlsProps = new KeyValueCollection();
tlsProps.addObject("tls", "1");
tlsProps.addObject("certificate", "client-cert.pem");
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using Map as a parameters source
Map<String, String> tlsProps = new HashMap<String, String>();
tlsProps.put("tls", "1");
tlsProps.put("certificate", "client-cert.pem");
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using Properties as a parameters source
Properties tlsProps = new Properties();
tlsProps.load(new FileInputStream("tls.properties"));
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using String as a parameters source
// Format corresponds to Transport Parameters as they appear in Configuration Manager
String tlsProps = "tls=1;certificate=client-cert.pem"; // No spaces around ";"
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);
```

Customizing TLS Configuration

When TLSConfiguration is prepared, it may still need some customization. Callback handlers for password retrieval, for example, cannot be configured in parameters and must be set explicitly. They should be set always, even if not used, because some security providers require them.

Specifying expected host name is not very straightforward and some aspects should be considered. When configuring TLS on client side, expected host names are in most cases different for primary and for backup connections. Though, on some virtualized environments, they can be the same. Users may choose to use IP addresses instead of DNS host names, or use DNS names with wildcards. Either way, expected host name must match one of names specified in server's certificate and in extreme cases it may not relate to actual host name at all. To account for these cases, setting expected host name is not automated in Platform SDK and left for user code. Example code below shows how to set this value to actual host name of target server.

According to X.509 specification, certificate may contain not just host name or IP address, but also URI or email address. Platform SDK supports only host names and IP addresses, but host name may use wildcard: a star symbol, "*", can be used instead of any one level of domain name.

Examples:

```
TLSConfiguration tlsConfiguration = ...;

// Applicable to both clients and servers
// Passwords are not used, so set dummies:
tlsConfiguration.setKeyStoreCallbackHandler(
        new DummyPasswordCallbackHandler());
tlsConfiguration.setTrustStoreCallbackHandler(
        new DummyPasswordCallbackHandler());

// In case some real password is needed:
tlsConfiguration.setKeyStoreCallbackHandler(new CallbackHandler() {
    public void handle(Callback[] callbacks) {
        char[] password = new char[] {
                'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};
            for (Callback c : callbacks) {
                if (c instanceof PasswordCallback) {
                    ((PasswordCallback) c).setPassword(password);
                }
            }
        }
    }
);

// Expected host name may contain exact host name, ...
tlsConfiguration.setExpectedHostname("someserver.ourdomain.com");
// wildcard host name, ...
tlsConfiguration.setExpectedHostname("*.ourdomain.com");
tlsConfiguration.setExpectedHostname("someserver.*.com");

// IPv4 address, ...
tlsConfiguration.setExpectedHostname("192.168.1.1");
// IPv6 address.
tlsConfiguration.setExpectedHostname("fe80::ffff:ffff:fffd");
```

Creating SSLContext

Platform SDK Commons has helper class – TLSConfigurationHelper, which creates SSLContext and SSLExtendOptions based on TLSConfiguration object. TLSConfigurationHelper has two methods:

```
public static SSLContext createSslContext(TLSConfiguration config);
```

and

```
static SSLExtendedOptions createSslExtendedOptions(TLSConfiguration config);
```

Method createSSLContext() determines security provider type if it is not set explicitly, creates necessary key store objects, key manager, trust manager, and finally wraps it all into SSLContext.

Method createSSLExtendedOptions() does not contain any logic, it just creates new SSLExtendedOptions with the exact parameters taken from TLSConfiguration.

Usage of both methods is shown in code sample below.

Example:

```
// TLS preparation section follows
KVList tlsProps = new KeyValueCollection();
tlsProps.addObject("tls", "1");
tlsProps.addObject("certificate", "client-cert.pem");
TLSConfiguration tlsConf =
        TLSConfigurationParser. parseClientTlsConfiguration(tlsProps);

boolean tlsEnabled = true;

SSLContext sslContext =
        TLSConfigurationHelper.createSslContext(tlsConfiguration);
SSLExtendedOptions sslOptions =
        TLSConfigurationHelper.createSslExtendedOptions(tlsConfiguration);

// The same as above, using shortcut methods:
sslContext = tlsConfiguration.createSslContext();
sslOptions = tlsConfiguration.createSslExtendedOptions();

Endpoint ep = new Endpoint(appName, host, port, null, tlsEnabled, sslContext, sslOptions);
```

TLSConfiguration Class

TLSConfiguration class is used as intermediate container to keep stronger-typed TLS parameters extracted from a parameter source. It contains the following:

Properties

TLSConfiguration Properties List

Name	Type	Description
tlsEnabled	boolean	Correspond to TLS parameters in Configuration; please see the list of TLS Parameters in Configuration Manager for details.
provider	String	
certificate	String	
certificateKey	String	
trustedCaCertificate	String	
mutual	boolean	
crl	String	
targetNameCheckEnabled	boolean	
cipherList	String	
fips140Enabled	boolean	
clientMode	boolean	Should be set to true for client-side of connection and false for server-side.

Name	Type	Description
		`TLSConfigurationParser` specialized methods set it automatically.
expectedHostname	String	Host name to check against, used when `targetNameCheckEnabled` is turned on. Typically is used by client side and assigned to the host/domain part of target URL.
keyStoreCallbackHandler	CallbackHandler	Please see Callback Handlers for details.
trustStoreCallbackHandler	CallbackHandler	
version	String	Defines security protocol version (all previous version will be accepted by default) Note: By default, the following client-side TLS Protocol versions are enabled in Java: • Java 6, Java 7 - TLSv1 • Java 8 - TLSv1.2
enabledProtocols	String	Limits supported security protocol version list (space separated list)
secProtocol	String Possible values are "SSLv23", "SSLv3", "TLSv1", "TLSv11", "TLSv12". Example: "sec-protocol=TLSv1"	Defines security protocol version (all previous version won't be accepted); available in Platform SDK from release 8.5.102.
keyStoreEntryCallbackHandler	CallbackHandler	It is used only for JKS provider. If it isn't assigned, then `trustStoreCallbackHandler` will be used as `keyStoreEntryCallbackHandler`.

Methods

TLSConfiguration Methods List

Signature	Description
SSLContext createSslContext()	A shortcut for `TLSConfigurationHelper.createSslContext` method. Creates and configures `SSLContext` object based on the properties values.
SSLExtendedOptions createSslExtendedOptions()	A shortcut for `TLSConfigurationHelper.createSslExtendedOptions` method. Creates `SSLExtendedOptions` object based on the properties values.

Constants

The following constants define supported values for a provider property:

- String TLS_PROVIDER_PEM_FILE;
- String TLS_PROVIDER_PKCS11;
- String TLS_PROVIDER_MSCAPI;

TLSConfigurationParser Class

TLSConfigurationParser class has methods that extract TLS parameters from different sources and create TLSConfiguration instance containing the parameters. It uses interface PropertyReader and several classes implementing this interface to read TLS parameters.

Methods

TLSConfiguration Methods List

Signature	Description
public static TLSConfiguration parseTlsConfiguration(final PropertyReader prop, final boolean clientMode)	This is the main and most generic method. It reads all possible TLS parameters (parameter names and possible values are detailed in the list of TLS Parameters in Configuration Manager), converts them and assigns them to `TLSConfiguration` properties.
public static TLSConfiguration parseServerTlsConfiguration(KVList kvl)	These methods provide shortcuts to parse TLS configuration from different source types.
public static TLSConfiguration parseClientTlsConfiguration(KVList kvl)	
public static TLSConfiguration parseServerTlsConfiguration(Map<String, String> map)	
public static TLSConfiguration parseClientTlsConfiguration(Map<String, String> map)	
public static TLSConfiguration parseServerTlsConfiguration(Properties prop)	
public static TLSConfiguration parseClientTlsConfiguration(Properties prop)	
public static TLSConfiguration parseServerTlsConfiguration(String transportParams)	
public static TLSConfiguration parseClientTlsConfiguration(String transportParams)	

Interface PropertyReader and Implementing Classes

Interface PropertyReader contains just one method:

```
String getProperty(String key)
```

Here, key argument contains name of parameter to extract. Implementing classes contain code that actually extract and return value corresponding to the key. Currently there are five implementations:

1. GConfigTlsPropertyReader - This class belongs to Application Template and is used to extract TLS parameters from a set of related Configuration objects. It cannot be included to Commons library since it would cause circular references between the Commons and Application Template.
2. KVListPropertyReader - Extracts String value from a KVList instance.
3. MapPropertyReader - Extracts value from a Map<String, String> instance.
4. PropertiesReader - Extracts value from a Properties instance.
5. TransportParamsPropertyReader - Parses transport parameters as they appear in Configuration Manager, for example:

```
“tls=1;certificate=c:/cert/cert.pem;mutual=1”.
```