## Security providers setup

While the code supports *BouncyCastle* and *EdDSA* security providers out-of-the-box, it also provides a way to [add security providers](https://issues.apache.org/jira/browse/SSHD-713) via the SecurityProviderRegistrar interface implementation. In order to add support for a new security provider one needs to implement the registrar interface and make the code aware of it.

### Default/built-in security provider registrars

The code contains built-in security provider registrars for *BouncyCastle* and *EdDSA* (a.k.a. ed25519). It automatically detects the existence of the required artifacts (since they are optional dependencies) and executes the respective security provider registration. This behavior is controlled by the org.apache.sshd.security.registrars system property. This property contains a comma-separated list of **fully-qualified** class names implementing the SecurityProviderRegistrar interface and assumed to contain a default **public** no-arguments constructor. The code automatically parses the list and attempts to instantiate and invoke the registrar.

**Note:**

* The registration code automatically parses the configured registrars list and instantiates them. In this context, one can use the special none value to indicate that the code should not attempt to automatically register the default providers.
* A registrar instance might be created but eventually discarded and not invoked if it is disabled, unsupported or already registered programmatically via SecurityUtils#registerSecurityProvider.
* The registration attempt is a **one-shot** deal - i.e., once the registrars list is parsed and successfully resolved, any modifications to the registered security providers must be done **programatically**. One can call SecurityUtils#isRegistrationCompleted() to find out if the registration phase has already been executed.
* The registrars are consulted in the same **order** as they were initially registered - either programmatically or via the system property configuration. Therefore, if two or more registrars support the same algorithm, then the earlier registered one will be used.
* If no matching registrar was found, then the default security provider is used. If none set, the JCE defaults are invoked. The default security provider can be configured either via the org.apache.sshd.security.defaultProvider system property or by programmatically invoking SecurityUtils#setDefaultProviderChoice. **Note:** if the system property option is used, then it is assumed to contain a security provider's **name** (rather than its Provider class name...).
* If programmatic selection of the default security provider choice is required, then the code flow must ensure that SecurityUtils#setDefaultProviderChoice is called before **any** security entity (e.g., ciphers, keys, etc...) are required. Theoretically, one could change the choice after ciphers have been been requested but before keys were generated (e.g....), but it is dangerous and may yield unpredictable behavior.

### Implementing a new security provider registrar

See AbstractSecurityProviderRegistrar helper class for a default implementation of most of the required functionality, as well as the existing implementations for *BouncyCastle* and *EdDSA* for examples of how to implement it. The most important issues to consider when adding such an implementation are:

* Try using reflection API to detect the existence of the registered provider class and/or instantiate it. The main reason for this recommendation is that it isolates the code from a direct dependency on the provider's classes and makes class loading issue less likely.
* Decide whether to use the provider's name or instance when creating security related entities such as ciphers, keys, etc... **Note:** the default preference is to use the provider name, thus registering via Security.addProvider call. In order to change that, either register the instance yourself or override the isNamedProviderUsed method. In this context, **cache** the generated Provider instance if the instance rather than the name is used. **Note:** using only the provider instance instead of the name is a rather new feature and has not been fully tested. It is possible though to decide and use it anyway as long as it can be configurably disabled.
* The default implementation provides fine-grained control over the declared supported security entities - ciphers, signatures, key generators, etc... By default, it is done via consulting a system property composed of org.apache.sshd.security.provider, followed by the security provider name and the relevant security entity - e.g., org.apache.sshd.security.provider.BC.KeyFactory is assumed to contain a comma-separated list of supported KeyFactory algorithms.

**Note:**

* The same naming convention can be used to enable/disable the registrar - even if supported - e.g., org.apache.sshd.security.provider.BC.enabled=false disables the *BouncyCastle* registrar.
* One can use all or \* to specify that all entities of the specified type are supported - e.g., org.apache.sshd.security.provider.BC.MessageDigest=all. In this context, one can override the getDefaultSecurityEntitySupportValue method if no fine-grained configuration is required per-entity type,
* The result of an isXxxSupported call is/should be **cached** (see AbstractSecurityProviderRegistrar).
* For ease of implementation, all support query calls are routed to the isSecurityEntitySupported method so that one can concentrate all the configuration in a single method. This is done for **convenience** reasons - the code will invoke the correct support query as per the type of entity it needs. E.g., if it needs a cipher, it will invoke isCipherSupported - which by default will invoke isSecurityEntitySupported with the Cipher class as its argument.
* Specifically for **ciphers** the argument to the support query contains a **transformation** (e.g., AES/CBC/NoPadding) so one should take that into account when parsing the input argument to decide which cipher is referenced - see SecurityProviderRegistrar.getEffectiveSecurityEntityName(Class<?>, String) helper method