GaraSign JCA/JCE User Guide

Version 1.25.0

Table of Contents

Preface

Document Information

Title	GaraSign JCA/JCE User Guide
Product Version	1.25.0

Trademarks

Disclaimer

Garantir makes no representations or warranties with respect to the contents of this document and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Furthermore, Garantir reserves the right to revise this publication and to make changes from time to time in the content hereof without the obligation upon Garantir to notify any person or organization of any such revisions or changes.

We have attempted to make these documents complete, accurate, and useful, but we cannot guarantee them to be without error or otherwise perfect. When we discover errors or omissions, or they are brought to our attention, we endeavor to correct them in succeeding releases of the product.

Please send any constructive comments on the contents of this document to the following email address: support@garantir.io

Overview

This document details how to install, configure, operate, troubleshoot, and uninstall the GaraSign Java Cryptography Architecture (JCA) Provider. This provider allows applications that integrate with the appropriate JCA APIs to sign and decrypt data using GaraSign. For more information on JCA, please consult the documentation for your specific Java installation.

Intended Audience

All products produced by Garantir are designed to be installed, configured, operated, and maintained by personnel with the necessary knowledge, skill, training, and qualifications to safely perform their duties. This document is intended for personnel responsible for installing, configuring, operating, and/or troubleshooting signing clients on Windows systems. It is assumed that the readers of this document and the users of its content are proficient with:

- Java
- Working from the command line
- Security concepts including, but not limited to, authentication, authorization, digital signatures, and logging
- Java tools such as jarsigner

Document Conventions

This document uses the following conventions to easily distinguish commands and alert you to important information.

Commands

Commands to be executed from a command prompt are provided in italics with a gray background, as shown below:

exampleCommand.exe /switch1

In some cases, commands will need to be edited with customer-specific values. These parts of the commands will be highlighted and in brackets, as shown below:

exampleCommand.exe /switch1 <insert some value>

Warnings

To help reduce the chance of data loss or corruption, this document provides warnings inside a red box with a warning icon, as shown below:

> **Warning**: Always exercise caution when performing security-related duties.

Components

This provider is made up of the following components:

1. Provider JAR Files – Provides functionality to integrate with GaraSign via JCA APIs.
2. Configuration File – Used to configure authentication, remote server information, etc.
3. Log Configuration (Optional) – Used to configure log settings and/or override the default logging implementation.

Supported Mechanisms

KeyStore

This provider supports authenticating to a GaraSign signing server and retrieving the user's authorized certificates and key handles. Note: this provider only receives identifiers to the private keys in GaraSign – it does **not** receive any of the private key's bytes.

Signature

This provider supports signing data using RSA or ECDSA with the following hash algorithms:

1. MD5 (Supported for legacy systems. Not recommended unless required.)
2. SHA-1 (Supported for legacy systems. Not recommended unless required.)
3. SHA-2-224
4. SHA-2-256
5. SHA-2-384
6. SHA-2-512
7. SHA-3-224

8. SHA-3-256
9. SHA-3-384
10. SHA-3-512

RSA-PSS is supported with the following hash algorithms as both the message hash and the mask generator function (MGF1):

1. SHA-1 (Supported for legacy systems. Not recommended unless required.)
2. SHA-2-224
3. SHA-2-256
4. SHA-2-384
5. SHA-2-512

Installation

Planning Your Installation

There are two ways to install/use this provider; choose the method that best suits your needs:

1. **Statically**

   Intended for systems that intend to use tools like *JarSigner* often.

2. **Runtime**

   Intended for developers that will access the provider programmatically via the JCA APIs.

Prerequisites

The following items are needed to install this JCE provider:

1. A computer with Java 8+ installed.
2. Permissions to modify the java.security file, if installing the provider statically.

Static Installation

To install the provider statically, execute the following steps:

1. Copy the GaraSignJceProvider.jar file to your Java classpath.
2. Edit the java.security file (located at *java.home/lib/security/java.security*) and add the following entry at the end of the list of currently configured providers:
   security.provider.#=com.garantir.jce.provider.GarantirJceSigningProvider
   Note: replace # with the appropriate number in the sequence

Runtime Installation

To install the provider for use at runtime, execute the following steps:

1. Copy the GaraSignJceProvider.jar files to your Java classpath.
2. At runtime, load the provider via the following line of code:
   Security.addProvider(new GarantirJceSigningProvider());

Configuration

Configuration of the provider is done by editing the properties file, located in a file of your choosing. Create a system property or an environment variable named *garantir_jce_provider_properties_file* and give it a value of the full path to your configuration file (file name and extension included). Note that the system property will take precedence over the environment variable so if both are set, the system property's value will be used. Edit the configuration file according to the table below. If neither the environment variable or system property are set, the provider will attempt to find and use the config.ini file from the GaraSign Windows, Linux, or macOS installer.

Configuration Values

Property Name	Property Value
base_url	[String] The base URL to the GaraSign signing server. For example, https://demo.garantir.io/CodeSigningRestService
issuer_url	[String] The OIDC Issuer URL. Required for OIDC authentication.
oidc_client_id	[String] The Client ID of the OIDC client (or app registration). Required for OIDC authentication via the Client Credentials flow.
oidc_client_secret	[String] The Client Secret of the OIDC client (or app registration). Required for OIDC authentication via the Client Credentials flow.
full_scope	[String] The OIDC scope to use. Required for OIDC authentication. Defaults to *openid*.
oidc_idp_type	[String] The type of OIDC identity provider. Required for OIDC authentication. Must be one of AZURE, OKTA, or GOOGLE. Defaults to AZURE.
username	[String] The username to authenticate with. Required for username/password authentication.
password	[String] The password to authenticate with. Required for username/password authentication.
client_keystore_location	[String] The location of the client keystore to use for client-certificate authentication. Required if using a software client keystore file.
client_keystore_password	[String] The password to the client keystore. Required for client-certificate authentication when using a software keystore file (e.g., JKS, PKCS12, etc.).
client_keystore_key_password	[String] The password to the client certificate's private key entry. If not specified, will default to the client_keystore_password value.
client_keystore_alias	[String] The alias of the client certificate entry in the client keystore.
client_keystore_type	[String] The type of keystore the client-certificate is in. If this value is set to any of the following values, client_keystore_location is not required to be set: - WINDOWS-MY - WINDOWS-ROOT - KEYCHAINSTORE

client_keystore_cert_hash	[String] The hex encoded hash of the certificate to use for client-certificate authentication. By default, it assumes the hash was computed using SHA-1. To support other hash functions, prefix the hash value with the hash function name followed by a colon. For example, SHA-256:adf3c6…
truststore_location	[String] Full file path to the trust store to use for TLS certificate chain validation. Defaults to *JAVA_HOME/lib/security/cacerts*.
truststore_password	[String] Password for the trust store. Defaults to *changeit*.
cache_file	[String] Full file path to the cache file to use for improved performance. If not specified, caching is not used.

Authentication

This provider currently supports the following authentication methods:

1. OpenID Connect (OIDC) via the Client Credentials Flow
2. Client-Certificate (optionally via Mutual TLS) for local users
3. Username / Password for local users

To configure method 1, set the *issuer_url*, *oidc_client_id, oidc_client_secret, full_scope*, and *oidc_idp_type* configuration values.

To configure method 2, set the appropriate *client_keystore_** configuration values.

To configure method 3, set the *username* and *password* configuration values.

Logging

Logging is done via slf4j. To allow for maximum flexibility for customers, this provider does not ship with a logging implementation. For customers unsure of which logging implementation to use, Garantir recommends using Logback. For more information on slf4j, please visit https://www.slf4j.org/.

Cache

To increase performance, this provider supports caching the session token so that each cryptographic function call does not need to perform its own authentication, but rather will share the session token from the first valid authentication. To enable session token caching, set the *cache_file* value in the configuration file.

> ⚠ **Warning**: While steps are taken to protect the cache, it should still be considered sensitive information. Users should take appropriate steps to safeguard their cache from unauthorized access.

Clearing the Cache

At times it may be necessary to clear the cache. There are two options to do this – manually and via the configuration file. To manually clear the cache, simple delete the cache file. To clear the cache via the configuration file, set the *cache_file* to a different value or comment it out to stop using the cache feature.

Usage

Once installed and configured, this provider can be used like any other JCA/JCE provider.

JarSigner

To use this provider with JarSigner, simply specify the appropriate parameters in your JarSigner command. If you have installed this provider statically, the command would be:

jarsigner -storetype GaraSign -storepass ignored -keypass ignored -keystore NONE -sigalg <insert signature algorithm> -signedjar <insert output file> <insert file to sign> <insert key name>

If you have installed this provider to be loaded at runtime, the command would be:

jarsigner -storetype GaraSign -providerClass com.garantir.jce.provider.GarantirJceSigningProvider -storepass ignored -keypass ignored -keystore NONE -sigalg <insert signature algorithm> -signedjar <insert output file> <insert file to sign> <insert key name>

Please note that the signature algorithm used must match the algorithm of the private key. For example, SHA256withECDSA would be appropriate for an EC key but would not work for an RSA key.

Programmatic Use

To use this provider programmatically, perform the following steps:

1. Load the provider (not necessary if the provider was installed statically)
2. Load the provider's key store
3. Retrieve the desired key from the key store
4. Use the provider's signature implementation along with the retrieve key to sign

Example:

```
Security.addProvider(new GarantirJceSigningProvider());
KeyStore keyStore = KeyStore.getInstance("GaraSign");
keyStore.load(null, null);
PrivateKey key = (PrivateKey) keyStore.getKey("<insert alias>", null);
Signature sig = Signature.getInstance("SHA256withRSA", "GarantirJceSigningProvider");
sig.initSign(key);
sig.update("<insert message to sign>".getBytes("UTF-8"));
byte[] signature = sig.sign();
```

Uninstall

To uninstall this provider, simply remove the jar files from the classpath. If this provider was installed statically, update the java.security file and remove the provider.

Frequently Asked Questions

Will this provider allow me to timestamp data that I sign?

Yes, when using this provider with tools like *jarsigner.exe*, you will still be able to timestamp data. This provider plays no role in the timestamping process as that is all handled by the calling application (in this example, *jarsigner*) and your TSA provider. In fact, this provider will not prohibit you from using any other features of *jarsigner* (or the other tools discussed in this document) as its functionality is only called when the actual data signing needs to occur.

What authentication methods does this provider support?

This provider supports both local and federated authentication methods. See the Authentication subsection of the Configuration section above for more details.

Am I still satisfying my requirement to use an HSM?

The short answer is yes. The GaraSign server that this provider communicates with sits in front of your company's cryptographic tokens. In most cases, these tokens are HSMs although GaraSign supports devices other than HSMs as well. Contact your GaraSign administrator to find out more about your organization's architecture and cryptographic tokens.

How can I tell if my data is signed?

This really depends on what you are signing. You can verify code signed by *jarsigner.exe* via the appropriate verification switches for the tool. Please consult the Java documentation on these tools for more information. Data signed programmatically can be verified using the *verify* method on the *Signature* object. Please note that you should use a third-party provider (e.g., Bouncy Castle) to verify signatures.

Troubleshooting

In general, most problems can be diagnosed by enabling logging and viewing the log file. See the Authentication

This provider currently supports the following authentication methods:

4. OpenID Connect (OIDC) via the Client Credentials Flow
5. Client-Certificate (optionally via Mutual TLS) for local users
6. Username / Password for local users

To configure method 1, set the *issuer_url*, *oidc_client_id, oidc_client_secret, full_scope*, and *oidc_idp_type* configuration values.

To configure method 2, set the appropriate *client_keystore_** configuration values.

To configure method 3, set the *username* and *password* configuration values.

Logging section for instructions on how to configure logging. Provided below is a list of common error messages one might see in an application, command prompt, or log file and some suggested ways to resolve each issue.

Cannot See a Key I was Given Access to

Description: A GaraSign administrator has granted an account access to a signing key, but the corresponding key does **not** show up in key store.

Cause: Caching is enabled and the current (non-expired) cache was created before access to the key was granted.

Resolution: Clear the cache and then reload the key store. See the Clearing the Cache section for more information.

Signing Takes Longer Than Expected

Description: Signing performance is not as good as witnessed when demoed by Garantir.

Cause 1: Caching is **not** enabled which results in a new authentication request on each signing request.

Resolution 1: Turn caching on to reduce the number of authentication requests made by the provider. See the Cache section for more information.

Cause 2: Logging is enabled which results in more file I/O on each signing request.

Resolution 2: Turn logging off and only enable it when you need to troubleshoot issues. See the Authentication

This provider currently supports the following authentication methods:

7. OpenID Connect (OIDC) via the Client Credentials Flow
8. Client-Certificate (optionally via Mutual TLS) for local users
9. Username / Password for local users

To configure method 1, set the *issuer_url*, *oidc_client_id, oidc_client_secret, full_scope*, and *oidc_idp_type* configuration values.

To configure method 2, set the appropriate *client_keystore_** configuration values.

To configure method 3, set the *username* and *password* configuration values.

Logging section for more information.

Cause 3: The calling tool (e.g., jarsigner) is being used in a verbose manner.

Resolution 3: Do not specify any *verbose* option when running the signing tool.

Cause 4: Not enough available resources. A non-trivial amount of the time spent processing is in calculating the hash locally and sending data over the network. These operations require system resources which may be currently allocated to other applications or processes. We have observed the mere presence of having certain browsers open or other applications running causes the signing request to take over three times as long as when those applications are closed.

Resolution 4: Close as many other applications and processes as possible before running your signing application. Also, use a computer with a fast CPU and sufficient RAM, whenever possible.

Cause 5: There is a large distance between the GaraSign server and the client.

Resolution 5: Ask your GaraSign administrator about deploying a GaraSign server closer to your client to reduce the distance data must travel over the network.

Cause 6: The GaraSign server that you are connecting to has been put into debug mode (most likely for troubleshooting). In debug mode, there is more file I/O per request which can degrade performance.

Resolution 6: Ask your GaraSign administrator if the GaraSign server is in debug mode. If it is, run another test once debug mode has been turned off.

Appendix A

Glossary of Terms, Abbreviations, and Acronyms

Certificate Authority (CA)	An entity that issues digital certificates.
Elliptic-Curve Diffie-Hellman (ECDH)	A variant of the Diffie-Hellman protocol that uses elliptic-curve cryptography.
Elliptic-Curve Digital Signature Algorithm (ECDSA)	A variant of the Digital Signature Algorithm which uses elliptic-curve cryptography.
Graphical User Interface (GUI)	A user interface that allows users to interact with it via visual indicators and graphical icons.
Hardware Security Module (HSM)	A physical computing device that safeguards, manages, and makes use of cryptographic keys.
Java Cryptography Architecture (JCA)	A provider-based architecture and set of APIs for producing digital signatures, message digests (hashes), certificates and certificate validation, encryption, etc.
Java Cryptography Extension (JCE)	An extension to the JCA that provides a framework and implementation for encryption, key generation, and more.
Message Digest 5 (MD5)	A legacy cryptographic hashing algorithm.
Optimal Asymmetric Encryption Padding (OAEP)	A padding scheme standardized in PKCS#1 v2.
Public Key Cryptography Standards (PKCS)	A group of public-key cryptography standards published by RSA Security LLC.
Public Key Infrastructure (PKI)	A set of roles, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption.
Registration Authority (RA)	The authority in a PKI that verifies requests for a digital certificate.
Rivest-Shamir-Adleman (RSA)	One of the first public-key cryptosystems.
Secure Hash Algorithm (SHA)	A set of cryptographic hashing with various output lengths.
Transport Layer Security (TLS)	A cryptographic protocol designed to provide communications security over a computer network.