Garantir

GaraSign Linux User Guide

Version 1.28.1

Table of Contents

Preface

Document Information

Title	GaraSign Linux User Guide
Product Version	1.28.0
Release Date	December 2024

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 client package for Linux. This client package allows users to integrate OpenSSL, GPG, and PKCS#11 based tools with GaraSign.

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 RPM signing clients. It is assumed that the readers of this document and the users of its content are proficient with:

- Linux
- Working from the command line
- Security concepts including, but not limited to, authentication, authorization, digital signatures, and logging
- Security tools such as GPG, OpenSSL, Java, and Cosign
- Linux package manager formats such as .rpm and .deb and their associated tools

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 -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 -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. PKCS#11 Library – Provides functionality to integrate with GaraSign via PKCS#11.
2. OpenSSL Engine – Provides functionality to integrate with GaraSign via OpenSSL.
3. Configuration File – Used to configure authentication, remote server information, etc.
4. GaraSign GPG Binaries – Customized GPG binaries to integrate with GaraSign.
5. Installation Package – Used to install and configure this provider.
6. GaraSign Command Line Package – Tools to list keys, assist with signing commands, export certificates, and more.

Installation

Planning Your Installation

This provider is supported on the following operating systems:

- CentOS 7.9
- CentOS 8
- CentOS 9
- Debian 10.13
- Debian 11.11
- Debian 12.7
- Ubuntu 18.04
- Ubuntu 20.04
- Ubuntu 22.04

- Ubuntu 24.04
- SUSE 15.1
- SUSE 15.2
- SUSE 15.3
- SUSE 15.4
- SUSE 15.5
- SUSE 15.6
- RHEL 7.9
- RHEL 8
- RHEL 9

Note: other Linux distributions that are based on any of the distributions listed above may additionally work, but they are not officially tested by Garantir unless explicitly stated otherwise for a specific integration or use case.

Prerequisites

The following items are needed to install this provider:

1. A computer running one of the supported operating systems listed above
2. A user account with sudo permissions
3. The appropriate provider bundle (GRS-GPG*) for the target operating system

Installation Steps

To install the provider, execute the following steps:

1. In an empty working folder, extract the provider bundle by executing the following command:
   tar -xvf <insert appropriate GRS.tar.gz file name>*
2. Install the package by executing the following command:
   sudo ./setup.sh
   Note: Optional parameters for the setup.sh script are:
   - *-p / --userpassword* - If set, it sets the user's password for local user authentication.
     Syntax:
     -p<password>, with no space between -p and <password>
     --userpassword=<password>, with '=' between --userpassword and <password>
   - *-k / --keepexistingglobalconf* – If specified, the existing global config.ini file, if present, will remain, otherwise it will be overwritten.
   - *-c / --createforallusers* – If specified, the user directories will be created for all users, instead of just for the user running the script.

   Note: If errors arise during installation, execute the following and then retry installation:
   CentOS:
   ```
   sudo yum search epel-release
   sudo yum info epel-release
   sudo yum install epel-release
   sudo yum config-manager --set-enabled PowerTools
   sudo yum remove tpm2-tss
   ```

   RHEL:

```
sudo yum search epel-release
sudo yum info epel-release
sudo yum install epel-release
sudo yum config-manager --set-enabled PowerTools
sudo yum remove tpm2-tss
```

Debian:
Ubuntu:
SUSE:
If a warning is displayed, enter ‘i’ to ignore and continue with installation.

3. Refresh the environment variables for your current working session by executing the following command:

```
source ~/.bashrc
```

Provider Configuration

Configuration of the provider is done by editing the INI configuration file, located at *$HOME/.config/Garantir/GRS/config.ini*. After a successful installation of the provider the default config.ini can be found in */etc/Garantir/GRS/config.ini* and a copy of it is made to each existing user’s *$HOME/.config/Garantir/GRS* directory, either during installation or when the user first uses the provider. If a new user account is created after this provider is installed, the default config.ini file will be copied over to the user’s *$HOME/.config/Garantir/GRS* directory when they invoke the provider, however, the other user directories and user environment variables will not be created for this user. In this case the user may want to execute the grsgpgconfig.sh or the grsgpgconfig_nosudo.sh script, where the latter script will only create the directories for the current user whereas the former will create the directories for all users on the system.

Configuration Values

The provided configuration file contains all the possible values and a description of each. Below is a list of the more commonly used values.

Property Name	Section	Property Value
Name	Server	[String] The DNS name or IP address of the GaraSign signing server load balancer. If multiple addresses are available, provide them as a comma separated list.
Port	Server	[int] The port number that the GaraSign signing server load balancer listens on.
URL	Server	[String] The path in the URL of the GaraSign signing server. Default=/CodeSigningRestService.
AlwaysUseGlobal	Config	[int] Set to 1 to always use the global config.ini instead of the user’s config.ini. Default = 0.
Enable	Cache	[int] Set to 1 to cache the session token. Set to 0 to not cache the session token and instead perform authentication on every cryptographic request. Default = 1.
TTL	Cache	[int] The time, in seconds, that the cache value is considered valid. Default = 300.

Enable	UI	[int] Set to 1 to have a UI dialog displayed for each signature operation. Set to 0 to not have any UI dialogs displayed. Default = 0.
Enable	Log	[int] Set to 1 to have log files generated. Set to 0 to not have log files generated. Default = 0.
Folder	Log	[String] The directory to write log files to. Note: this directory must already exist and the user must have write permissions to it.
Type	OpenID	[String] Specifies the type of OIDC identity provider being used. Supported values are AZURE, OKTA, and GOOGLE.
OfflineMode	OpenID	[int] Set to 1 to have refresh tokens be used. Set to 0 to not use refresh tokens which will result in a new authentication flow each time the GaraSign session expires. Default = 0.
IssuerURL	OpenID	[String] The Issuer URL for the OIDC identity provider.
GRSClientAppID	OpenID	[String] The client application ID for OIDC authentication.
FullScope	OpenID	[String] In most cases this should be set to *openid*. For legacy use cases, it should be set to *api://<GRSWebAPIAppID>/<scope>*, where <scope> is the actual scope of the GRS REST Web API App (e.g access_as_user).
GRSClientAppSecret	OpenID	[String] The client secret when using the OIDC Client Credentials Flow with a client secret.
GRSClientAppP12File	OpenID	[String] The PKCS12 file containing the client private key when using the OIDC Client Credentials flow with key-based authentication.
GRSClientAppP12Pass	OpenID	[String] The password to the PKCS12 file containing the client private key when using the OIDC Client Credentials flow with key-based authentication.
GRSClientAppCertificateFile	OpenID	[String] The PEM file containing the client certificate when using the OIDC Client Credentials flow with key-based authentication.
GRSClientAppKeyFile	OpenID	[String] The PEM file containing the client private key when using the OIDC Client Credentials flow with key-based authentication.
Count	Users	[int] For future functionality. Leave as 1 for now.
Authentication	User1	[String] The authentication method to use. Options are *Kerberos* and *Normal*. Use *Kerberos* to enable Kerberos Authentication (via SPNEGO). Use *Normal* to enable certificate authentication (i.e., mutual TLS) or username and password authentication.
AllowGlobalKerberos	User1	[int] Set to 0 to restrict Kerberos authentication to intranet sites. Set to 1 to enable Kerberos authentication to all sites, including those on the internet. Default = 0.
Username	User1	[String] The username to authenticate with if authenticating with username and password.
Password	User1	[String] The password to authenticate with if authenticating with username and password.

P12File	User1	[String] The absolute path to the PKCS#12 file to use for client-certificate authentication. Note: **not** compatible with CentOS
P12Pass	User1	[String] The password to the PKCS#12 file to use for client-certificate authentication. Note: **not** compatible with CentOS
CertificateFile	User1	[String] The absolute path to the PEM encoded certificate to use for client-certificate authentication.
KeyFile	User1	[String] The absolute path to the PEM encoded private key file to use for client-certificate authentication.

Authentication

This provider supports Kerberos, OpenID Connect (OIDC), Username & Password, and Client Certificate authentication. Kerberos and OIDC are supported for federated users from supported identity providers whereas Username & Password and Client-Certificate authentication are supported for local users.

Kerberos

To enable Kerberos authentication, set the *Authentication* value to *Kerberos.* You can optionally choose to restrict Kerberos authentication to intranet sites or open it up to non-intranet sites by setting the *AllowGlobalKerberos* value. By default, Kerberos authentication is restricted to intranet sites only.

Kerberos Example

```
[User1]

Authentication=Kerberos
```

Username & Password

To enable username & password authentication, set the *Authentication* value to *Normal* and set the *Username* and *Password* values to the ones provided by your GaraSign system administrator in the *User1* section of your user's configuration file. Note: the username & password values are **not** the same as your operating system login credentials so do **not** place those values in this configuration file.

Username & Password Example

```
[User1]

Authentication=Normal

Username=signing_client_username

Password=$ign_Ev3ry+h1nG!
```

Client Certificate

To enable client certificate authentication, set the *Authentication* value to *Normal* and set either the *P12File* and *P12Pass* values or the *CertificateFile* and *KeyFile* values in the configuration file.

Note: The CentOS provider does **not** support the *P12File* or *P12Pass* format of configuration.

Client Certificate Example (PKCS#12)

```
[User1]
```

```
Authentication=Normal

P12File=/absolute/path/to/clientAuth.p12

P12Pass=My_Pl2_P@s5!
```

Client Certificate Example (PEM)

```
[User1]

Authentication=Normal

CertificateFile=/absolute/path/to/cert_clientAuth.pem

KeyFile=/absolute/path/to/key_clientAuth.pem
```

Logging

To enable logging, set the *Enable* property to *1* and the *Folder* property to the desired directory in the *Log* section of the configuration file.

Note: The configured logging folder must exist and be writable by the process that will utilize the provider.

⚠ **Warning**: Logging degrades performance, so it is recommended to only enable it to troubleshoot issues.

Logging Example

```
[Log]

Enable=1

Folder=/var/tmp
```

Cache

To increase performance, this provider supports caching the session token so that each cryptographic request 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 *Enable* property to *1* and the *TTL* value to the desired time, in seconds, that the session token should be cached for in the *Cache* section of the provider's configuration file.

Note: The maximum time that the session token can be valid for is 7 hours and 59 minutes = 28740 seconds

⚠ **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 TTL value. To manually clear the cache, delete the contents of the cache folder located at *$HOME/.config/Garantir/GRS*. To clear the cache via the TTL value, set the TTL value to something very small (e.g., 1).

OpenSSL Configuration

GaraSign can be used with OpenSSL via the GaraSign OpenSSL Engine. In many cases, GaraSign can be integrated with the OpenSSL that is installed on the host system, but there are cases where this is not possible and so GaraSign comes with its own OpenSSL located at */opt/Garantir/openssl-1.0.2/bin/openssl*. Once the provider is configured, create an OpenSSL configuration file in a location of your choosing with the following content:

```
openssl_conf = openssl_init
# ---------------------------
# GaraSign
# ---------------------------
[openssl_init]
engines = engine_section

[engine_section]
# Configure ENGINE named "grs"
grs = grs_section

[grs_section]
# grs ENGINE specific commands
engine_id = grs
dynamic_path = /opt/Garantir/lib/libgrsengine_1.0.2.so
# or use /opt/Garantir/lib/libgrsengine.so
default_algorithms = ALL
init = 1
# ---------------------------
```

Then link to the configuration file via the OPENSSL_CONF environment variable by setting its value to the path to the configuration file. After that, GaraSign can be used via standard OpenSSL Engine integration. Be sure to unset the environment variable after use if you no longer wish to integrate with the GaraSign OpenSSL Engine.

Listing Keys

To list keys with the OpenSSL that is installed on the client machine, execute the following command:

```
openssl engine grs -post LIST_KEYS_STDOUT -t -c
```

To list keys with the GaraSign-provided OpenSSL, execute the following command:

```
/opt/Garantir/openssl-1.0.2/bin/openssl engine grs -post LIST_KEYS_STDOUT -t -c
```

Export Keys

Some tools require a PEM private key file as input. While GaraSign does **not** export the actual private key bytes, it supports generating a PEM file to satisfy these tools using fake data as the private key. To export such a file with the OpenSSL that is installed on the client machine, execute the following command:

```
openssl engine grs -post EXPORT_KEY_AND_CERT:<keyname>:<obfuscated_key_and_cert_absolute_path> -t -c
```

To export such a file with the GaraSign-provided OpenSSL, execute the following command:

/opt/Garantir/openssl-1.0.2/bin/openssl engine grs -post EXPORT_KEY_AND_CERT:<keyname>:<obfuscated_key_and_cert_absolute_path> -t -c

GPG Configuration

GPG can be configured automatically or manually. Garantir strongly recommends using automatic configuration.

GPG Automatic Configuration

Once the provider is configured, configure GPG for signing by executing the following command with your regular user account:

/opt/Garantir/bin/GrsGPGLoader

Upon successful completion, the user's GPG configuration will be set in *$HOME/.gnupggrs*.

GPG Manual Configuration

⚠ **Warning**: Garantir strongly recommends against manual configuration. This approach will result in signature mismatches when signing on different client machines, even when using the same server-side key.

Once the provider is configured, configure GPG for signing by performing the following steps with your regular user account:

1. Start the GPG agent by executing the following command:
   /opt/Garantir/bin/gpg-agent --daemon --homedir $HOME/.gnupggrs/ --options $HOME/.gnupggrs/gpg-agent.conf
   Note: The GPG agent *must* always be started *before* attempting to sign with this provider.
2. Read the contents of the PKCS#11 module by executing the following command:
   /opt/Garantir/bin/gpg --homedir $HOME/.gnupggrs/ --card-status
3. Retrieve the available keygrips by executing the following command:
   echo "SCD LEARN" | /opt/Garantir/bin/gpg-agent --homedir $HOME/.gnupggrs/ --server --options $HOME/.gnupggrs/gpg-agent.conf /opt/Garantir/bin/gpg-connect-agent | grep KEY-FRIEDNLY
4. Copy the keygrip (i.e., hexadecimal value) of the desired key
5. Edit *$HOME/.gnupggrs/gnupg-pkcs11-scd.conf* in a text editor and add the following line to the end of the file: openpgp-sign <insert key grip>
   Example: *openpgp-sign E6601EC66DFFF252ADF84E3EEF82FFE5D80406BD*
6. Provide a mapping of the GaraSign key to your GPG configuration by executing the following command:
   /opt/Garantir/bin/gpg --homedir $HOME/.gnupggrs/ --expert --full-generate-key
   a. Choose option 14 for Existing Key from card
   b. Choose the appropriate key
   c. Toggle the capabilities so that only Certify and Sign are selected
   d. Enter the name to bind this key to
      Note: Remember this value as it will be needed in a future step
   e. Enter the email address to bind this key to
      Note: Remember this value as it will be needed in a future step
   f. Complete the rest of the prompts as desired
7. Verify the new key has been added to your keyring by executing the following command:
   /opt/Garantir/bin/gpg --homedir $HOME/.gnupggrs/ --list-secret-keys --keyid-format LONG

8. Export the public key from your keyring by executing the following command:
   `/opt/Garantir/bin/gpg --homedir $HOME/.gnupggrs/ --export -a <insert email_address> > mykey.asc`

Usage

Once installed and configured, the GPG binary located at */opt/Garantir/bin/gpg* can be used as a replacement for signing commands that call *gpg*. For example, the standard GPG command of

```
gpg --output <insert output file name> --sign <insert file to sign>
```

would become

```
/opt/Garantir/bin/gpg --homedir $HOME/.gnupggrs --output <insert output file name>--sign <insert file to sign>
```

Tools that make use of GPG for signing can also make use of the GaraSign GPG binaries by configuring the tools appropriately.

Signing RPM Packages

Signing with RPM requires rpm and rpm-sign to be installed. Execute the following command to install these components if they are not already installed on the client machine:

CentOS:	*sudo yum install rpm-sign*
RHEL:	*sudo yum install rpm-sign*
Debian:	*sudo apt install rpm*
Ubuntu:	*sudo apt install rpm*
openSUSE:	*sudo zypper install rpm*

If the file *$HOME/.rpmmacros* doesn't exist yet, create it and add the following content to it:

```
%_signature gpg

%_gpg_path /home/<insert user>/.gnupggrs

%__gpg /opt/Garantir/bin/gpg

%_gpg_name <insert name used during configuration>
```

RPM packages can now be signed by executing the following command:

```
rpm --addsign <insert path to .rpm file>
```

If prompted for a passphrase, press Enter. Note: for automated environments tools like *expect* can be used to automatically respond to the passphrase prompt.

To verify an RPM package, execute the following steps:

1. Import the public key into the RPM database by executing the following command:
   `sudo rpm --import <insert path to exported public key>`
   Note: This command might not work on Debian or Ubuntu
2. Verify the signature by executing the following command:
   `rpm --checksig --verbose <insert path to signed .rpm file>`

Note: This command might not work on Debian or Ubuntu

Java

GaraSign can integrate with Java-based tools and custom code through multiple means. On supported Linux platforms, the recommended approach is to use GaraSign's PKCS#11 provider (please see the GaraSign JCA/JCE User Guide for details on a pure Java integration). To integrate with PKCS#11, create a configuration file (e.g., grspkcs11.conf) and place the following content in it:

```
name = GaraSign
```

```
library = /usr/local/lib/Garantir/GRS/libgrsp11.so
```

Then pass the configuration file to the SunPKCS11 provider and load the keys through the JCE's KeyStore.

JarSigner

Pass the PKCS#11 configuration file to JarSigner while specifying the SunPKCS11 provider. For example:

```
jarsigner -keystore NONE -storetype PKCS11 -providerClass sun.security.pkcs11.SunPKCS11 -providerArg
"/path/to/grspkcs11.conf" -storepass ignored -sigalg SHA256withRSA <file to sign> <key name>
```

Note: Timestamping can be added via the -tsa switch in JarSigner. Garantir strongly recommends that customers timestamp their code so that signatures remain valid past the expiration of the signing certificate.

Programmatic Use

To use the SunPKCS#11 integrated with GaraSign 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 – Dynamic Provider instantiation:

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.PrintStream;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.nio.charset.StandardCharsets;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.Provider;
import java.security.Security;
import java.security.Signature;

public class Main {

  public static void main(String[] args) throws Exception {
         // load provider
      ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
      PrintStream ps = new PrintStream(byteStream);
      String providerName = "SunPKCS11-GRS";
      ps.println("name = " + providerName);
      String libPath = sanitizePath("/usr/local/lib/Garantir/GRS/libgrsp11.so");
      ps.println("library = \"" + libPath + "\"");
      Provider pkcs11Provider = getProvider(byteStream.toByteArray());
      Security.addProvider(pkcs11Provider);

      // load GaraSign PKCS11 keystore
      KeyStore ks = KeyStore.getInstance("PKCS11");
      ks.load(null, null);

      // retrieve desired private key
      String keyName = args[0];
      PrivateKey privKey = (PrivateKey)ks.getKey(keyName, null);

      // sign data with key
      String algorithm = args[1]; // examples: SHA256withRSA, SHA256withECDSA, etc.
      Signature sig = Signature.getInstance(algorithm, pkcs11Provider);
      sig.initSign(privKey);
      sig.update("hello, world".getBytes(StandardCharsets.UTF_8));
      byte[] signature = sig.sign();
  }

  private static Provider getProvider(byte[] configurationInfo) throws Exception {
    Provider PROV;
    String pkcs11Config = new String(configurationInfo, "UTF-8");

    try {
            // newer methods of Java use this approach
      Method configure = Provider.class.getMethod("configure", new Class[] {
String.class });

      PROV = Security.getProvider("SunPKCS11");
      PROV = (Provider)configure.invoke(PROV, new Object[] { pkcs11Config });

    } catch (NoSuchMethodException e) {
            // fall back for older methods of Java
      Constructor<?> SunPKCS11 =
Class.forName("sun.security.pkcs11.SunPKCS11").getConstructor(new Class[] {
InputStream.class });
      ByteArrayInputStream confStream = new
ByteArrayInputStream(pkcs11Config.getBytes());
      PROV = (Provider)SunPKCS11.newInstance(new Object[] { confStream });
    }

    return PROV;
  }

  private static String sanitizePath(String path) {
      if (path == null) {
        return null;
      }

      return path.replace("\\", "\\\\");
  }

}
```

Container Image Signing

GaraSign supports multiple methods of signing container images. On Linux, the recommended approach is to use cosign (from the Sigstore project) integrated with GaraSign's PKCS#11 provider. To do this, first list the tokens via the following command:

```
cosign pkcs11-tool list-tokens --module-path "/usr/local/lib/Garantir/GRS/libgrsp11.so"
```

Retrieve the slot number (there should onle be one) and then execute the following command:

```
cosign pkcs11-tool list-keys-uris --module-path "/usr/local/lib/Garantir/GRS/libgrsp11.so" --slot-id 1 --pin 1234"
```

Retrieve the URI for the desired key and then execute the following command:

```
cosign sign --key "<PKCS11 URI>" <image to sign>
```

VMware Image Signing

GaraSign supports signing Open Virtualization Format (OVF) files via ovftool. To use this integration, first export the obfuscated private key in PEM format as described in the OpenSSL section above. After that, execute the following command to sign:

```
ovftool --privateKey=<obfuscated_key_and_cert_absolute_path> <input_file> <signed_file>
```

After signing, you can inspect the contents by untarring the signed .ova file (e.g., tar -tf img-signed.ova). To verify the signed file, place the CERT file in the same directory as the signed .ova file and then execute the following command:

```
ovftool <signed file>
```

If the signature validates successfully, the command presents its metadata. Otherwise, it will display a message about the inability to validate the signature.

Signing Git Commits

To use this provider to sign git commits, execute the following steps:

5. Navigate to the directory of the desired git repository
6. Instruct git to use the desired key to sign commits in this repository by executing the following command:
   `git config --local user.signingkey <insert GPG Key ID>`
7. Instruct git to use GaraSign's GPG binary by executing the following command:
   `git config --local gpg.program /opt/Garantir/bin/gpg`
8. Instruct git to use the correct username and email for the commit by executing the following commands:
   `git config --local user.name <insert name>`
   `git config --local user.email <insert email>`
9. Sign your commit by executing the following command:
   `git commit -S -m "<insert message>"`

To check that a commit has been signed successfully, execute the following command: *git log --show-signature*

Secure Shell (SSH)

To use this provider with SSH, execute the following steps:

1. If the ssh-agent is not already running, start it with the following command:
   ```
   eval `ssh-agent -s`
   ```
2. Add the GaraSign keys to the ssh-agent with the following command:
   ```
   ssh-add -s /usr/local/lib/Garantir/GRS/libgrsp11.so
   ```
3. Optionally, list the GaraSign keys that have been added to the agent with the following command:
   ```
   ssh-add -L
   ```
4. Connect to the desired SSH host using standard SSH commands such as:
   ```
   ssh <insert username>@ <insert host>
   ```

Snowflake

Applications accessing Snowflake via JDBC or ODBC may be able to use GaraSign keys for key-pair authentication, sometimes referred to as JWT authentication by some applications.

ODBC

The Snowflake ODBC driver v3.0.1+ links with OpenSSL 3.x and can therefore make use of the GaraSign OpenSSL 3.x Provider. If the calling application provides its own *openssl.cnf* file, then that file must be modified to add the GaraSign OpenSSL Provider at the top (be sure to backup the file before modifying it). If the solution does not provide its own *openssl.cnf* file, then one must be created with the GaraSign OpenSSL Provider configured and an environment variable named *OPENSSL_CONF* must be created that points to the newly created *openssl.cnf* file (be sure to create the environment variable such that the application can read it at runtime).

Once the OpenSSL configuration file and environment variable are both set, export an obfuscated key (this will not contain the real private key bytes) as described in the sections above and demonstrated in the example below:

```
export OPENSSL_CONF= /path/to/openssl.cnf
openssl pkey -in name:<insert key name> -out /path/to/output/file
```

The final step is to modify the ODBC connection string used by the application to point the *PRIV_KEY_FILE* attribute to the path of the obfuscated key file (i.e., the output of the *openssl pkey* command above).

Python

Python applications can leverage GaraSign's PKCS#11 library via the GaraSign-provided *PKCS11RSAPrivateKey* class defined in *pkcs11_rsa.py*. Once imported, the applications use this class as shown below:

```python
import snowflake.connector as sc
from pkcs11_rsa import PKCS11RSAPrivateKey
 with PKCS11RSAPrivateKey('Garantir Token', '<insert GaraSign key name>') as private_key:
   conn_params = {
      'account': '<insert account number>',
      'user': '<insert username>',
      'private_key': private_key,
```

```
    'database': '<insert database>',
    'schema': '<insert schema>'
}
 conn = sc.connect(**conn_params)
cs = conn.cursor()
cs.execute("SELECT CURRENT_VERSION()")
one_row = cs.fetchone()
print(one_row[0])
cs.close()
conn.close()
```

JDBC

Java applications that connect to Snowflake via JDBC can make use of either the PKCS#11 provider or pure Java JCE provider. This document describes the PKCS#11 provider approach but due to some limitations on the Snowflake driver end users may wish to use the pure Java JCE provider which is documented in its own user guide.

Perform the same steps as described in the Java section above, but additionally set three environment variables, some additional SunPKCS11 configuration, and ensure that the GaraSign PKCS#11 library is the first configured security provider.

Environment Variables:

```
GRS_PKCS11_FORCE_PRIVKEYS_NOT_SENSITIVE=1
GRS_PKCS11_FORCE_PRIVKEYS_EXTRACTABLE=1
GRS_PKCS11_FORCE_DUMMY_PRIVKEYS_MATERIAL=1
GRS_PKCS11_FORCE_CREATED_OBJECTS_ON_TOKEN=1
```

Additional SunPKCS11 Configuration:

Specify the following when configuring the SunPKCS11 provider:

```
ps.println("disabledMechanisms = {");
ps.println("    CKM_AES_CBC");
ps.println("    CKM_AES_CBC_PAD");
ps.println("    CKM_AES_KEY_GEN");
ps.println("    CKM_ECDH1_DERIVE");
ps.println("    CKM_MD5");
ps.println("    CKM_SHA_1");
ps.println("    CKM_SHA256");
ps.println("    CKM_SHA384");
ps.println("    CKM_SHA512");
ps.println("    CKM_RSA_PKCS_KEY_PAIR_GEN");
ps.println("    CKM_EC_KEY_PAIR_GEN");
ps.println("}");
```

Ensure GaraSign is the 1st Provider:

```
Security.insertProviderAt(pkcs11Provider, 1);
```

Use the GaraSign Key

Once instantiated, retrieve the desired key and pass it to the connection properties as “privateKey”, as shown below:

```
prop.put("privateKey", getPrivateKey());
Connection conn = DriverManager.getConnection(url, prop);
```

Frequently Asked Questions

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.

Do I need a Certificate Authority to sign with GPG?

No, GPG uses the web of trust model which does not require Certificate Authorities.

How can I tell if my data is signed?

This really depends on what you are signing. You can verify RPM packages signed by *rpm* via the appropriate verification switches for the tool. Please consult the RPM documentation for more information.

How do I stop being prompted for a passphrase when signing RPM packages?

You can use other tools like *expect* to automate the process of pressing Enter at the passphrase prompt.

Note: Garantir makes no warranty on any third party tool, including *expect*.

Troubleshooting

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

Unable to Sign

Description: A signing or related command fails with an obscure error message.

Cause 1: The configuration is incorrect.

Resolution 1: Check the values in your configuration file (*$HOME/.config/Garantir/GRS/config.ini*) and ensure their validity.

Cause 2: The GPG agent is not running.

Resolution 2: Start the GPG agent and try signing again. See the GPG configuration section for more information.

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 as available.

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 logging section for more details.

Cause 3: The calling tool (e.g., *rpm*) 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 lots of RAM, whenever possible.

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

Resolution 5: Ask your GaraSign administrator about standing up 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 when 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.
GNU Privacy Guard (GPG)	A software implementation of RFC 4880.
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.
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.
RPM Package Manager (RPM)	Open source package manager system used on some Linux systems.
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.