SafeNet Authentication Service (SAS)

Java Authentication API Developer's Guide

gemalto
security to be free

Document Part Number: 007-012400-002, Rev. F

Release Date: June 2016

Contents

Overview

The SafeNet Authentication Service (SAS) Authentication API enables agents to support all functionality required to interact with the SAS authentication server. SAS agents are third-party applications with embedded plug-in code that enables the collection of user names and one-time passwords (OTPs) to be passed to the SAS server for verification.

SAS Authentication API

The SAS Authentication API is represented by a single Java class, **CRYPTOCardAPI**. This class is a singleton.

Constructors

```
CRYPTOCardAPI()
{
      // empty constructor
}
```

getInstance Method

```
getInstance
(
)
```

returns a singleton instance of the `CRYPTOCardAPI` class.

setINIPath Function

```
setINIPath
(
      String path
);
```

This function may be called to set the location from where the API should load the INI file. If this function is not called before calling LoadJNILibrary, the INI file will be loaded from:

- Default Deployment Location
- If not persent at default location, the current execution path will be explored.

LoadJNILibrary must be called to initialize this instance.

LoadJNILibrary Method

```
LoadJNILibrary
(
)
```

This functions loads and initializes the SafeNet encryption, communication, and logging library from the system-exported **LD_LIBRARY_PATH** (on Linux) environment path or the system **PATH** (on Windows).

This function requires that, on Linux-like systems, the **LD_LIBRARY_PATH** environment variable be exported, pointing to the location of all the binaries like **libJCryptoWrapper.so**. On Linux-like systemd, run **ldd ./libJCryptoWrapper.so** to confirm that all the dependencies can be resolved.

On Windows Systems, the management tool adds this path to the system's **PATH** environment variable, making it visible to the Java application. But if it is desired to load binaries from a different location (such as multiple applications using their own binaries), this function should be called with the full path to **JCryptoWrapper.dll** or **libJCryptoWrapper.so**.

On a Linux system, you must add the **LD_LIBRARY_PATH** export directive in your application`s launch script. In the case of a web application, this must be declared in the web server launch script.

Read and write permissions must be set on the defined paths before attempting to load the binaries.

```
LoadJNILibrary
(
      String LoadFromLocation
);
```

This functions loads and initializes the SafeNet encryption, communication, and logging library from the required location. Call the **setINIPath** function before calling this function if it is desired to load the INI file from any other location than the default. Comments for the **LoadJNILibrary** function without any argument are also valid for this function call.

Authenticate Method

All authentications related actions make use of a single API call:

```
Authenticate
(
      String[] arrData
);
```

Where the **arrData** details are:

arrData[0]	UserName	In Value
arrData[1]	Organization	In Value
arrData[2]	OTP	In Value
arrData[3]	Challenge	Return Value

arrData[4]	State	In and Out
arrData[5]	ChallengeData	Return Value
arrData[6]	ChallengeMessage	Return Value
arrData[7]	ReturnedResult	Return Value
arrData[8]	BothServersDown	Return Value -> 1 or 0 (1 if down)
arrData[9]	ErrorMessage	Return Value -> Error Message for Logging or client
arrData[10]	InIPAddress	In Value Value -> Incoming client IP address - Service Provider IP address

Where:

- Passing NULL as parameters should be avoided. Instead, array must be initialized with empty strings.
- UserName – A string representing the user name of the individual who is authenticating.
- Organization – A string representing the organization to which the individual who is authenticating belongs. This currently should be passed as an empty string to represent the default organization.
- OTP – A string representing the user's passcode. This may take form of either:
  - [PIN+OTP] – Server-side PIN authentication.
  - [OTP] – Token-side PINs or no PIN.
  - [PIN] – When responding to a server-side user changeable PIN change request.
  - [StaticPassword] – User has a static password enabled or is responding to a static password change.

  This parameter may also be set to **null** to indicate a challenge is required.
- Challenge – A string that may be populated with a challenge/PIN change/outer window authentication message.
- State – A string that may be populated with a **state** attribute. When returning a challenge, the same state should passed back to the server.
- ChallengeData – Data returned as the response to the challenge.
- ChallengeMessage – Returned user message appended with the challenge.
- ReturnedResult – Returned result (String):
  - 0 – Authentication Failed
  - 1 – Authentication Succeeded
  - 2 – Challenge
  - 3 – Server provided PIN
  - 4 – User needs to provide PIN
  - 5 – Authentication in outer window; Re-authenticate
  - 6 – User must change their static password
  - 7 – Static password change does not satisfy policies
  - 8 – PIN provided doesn't meet requirements. Please provide a new PIN

- BothServersDown – TokenValidator servers are down. If both servers (primary and secondary) are down, value is 1, otherwise 0.
- InIPAddress – A string representing the IP address from which the authentication request came. If this parameter is an empty string, the SAS server will attempt to detect the IP from which the authentication request came from.

VerifySignature Method

Verifies the token's signature for a given hash.

```
VerifySignature
(
String[] arrData
);
```

where arrData details are:

arrData[0]	SerialNumber	In Value
arrData[1]	Hash	In Value
arrData[2]	Signature	In Value
arrData[3]	ReturndResult	Return Value

Return Value could be:

- 0 – Signature is incorrect for hash provided.
- 1 – Signature is correct for hash provided.

where:

- Passing **NULL** as a parameter should be avoided. Instead, an array must be initialized with empty strings.
- SerialNumber – A string representing the token's serial number.
- Hash – A string representing the hash value to verify.
- Signature – A string representing the signature to verify for the provided hash.
- ReturndResult – A string return value: 1 = Success, 0 = Failure.

getGridSureGrid Method

The **getGridSureGrid** method creates and returns a GrIDsure grid from the received SAS challenge.

```
getGridSureGrid
(
      String BSIDChallenge
);
```

The **getGridSureGrid** method returns an instance of the **BufferedImage** class (bitmap).

getGridSureLogo Method

```
getGridSureLogo
(
);
```

The **getGridSureLogo** method returns the GrIDsure logo (an instance of the **BufferedImage** class) from a string saved in this function.

If the **getGridSureLogo** method fails, it returns **NULL BufferedImage** or throws an exception.

Use Case Scenarios

SAS architecture supports the concept of a token-side and server-side PIN used in either QuickLog or challenge/response mode. In addition, agents must support challenges, inner/outer window authentication, and static password authentication. The following sections discuss these features in more detail.

Basic Authentication

The communication between an SAS agent and server is similar to that used in the RADIUS protocol, in that the concepts of challenge messages and states are used.

The following scenario shows the most basic interaction between the agent and server:

1. Authenticating agent issues an authentication request with username, organization name, and a passcode.
2. Server responds with one of nine possible return codes as outlined in **Returned Result**.

Challenge-Response

Central to the concept of challenge/response authentication, outer window authentication, and server-side PIN changes is a challenge message and state attribute issued from the authenticating server. This mechanism is employed to authenticate tokens in challenge/response mode in the following manner:

1. Authenticating agent issues an authentication request with username, organization name, and a blank passcode.
2. Server responds with a challenge message containing a challenge string (for example, **challenge: 19863257**) and a **state** attribute.
3. The authenticating agent responds to the challenge by issuing another authentication request with their username, organization name, a response, and the state attribute.

To support localization, the SAS server returns necessary data only in its challenge messages, and the agent is required to construct a localized version of it to display to the client. For example, the SAS server would return only **19863257**, and the SAS agent would display “Please respond to the challenge: 19863257”.

Outer Window Authentication

The concept of challenge messages and maintaining state is also used when authenticating a user through inner/outer window authentication. Outer window authentication allows a user to authenticate by providing a match in a large look-ahead window; however, they must also be able to respond to a follow-up challenge that asks them to provide the exact next OTP from their token. The following sequence illustrates this functionality:

1. The authenticating agent issues an authentication request with username, organization name, and a passcode.
2. The server finds a match for the provided OTP in the outer window and issues a challenge to the client containing an outer window authentication string (“Please re-authenticate using the next OTP from your token”) and a **state** attribute.
3. The authenticating agent responds to the challenge by issuing another authentication request with their username, organization name, a response, and the state attribute.

See the localization note in the section “Challenge-Response” page 9.

PIN Styles

A number of PIN styles in SAS are supported:

- No PIN
- Fixed PIN (token-side PIN validation)
- User-changeable PINs. (token-side PIN validation)
- Stored on server, fixed PIN
- Stored on server, user-changeable PIN
- Stored on server, server-changeable PIN

The authentication mechanism in SAS supports incoming passcodes in the following format [PIN+OTP], [OTP], [NEWPIN], [STATICPASSWORD] and an empty passcode to request a challenge.

In the case of PINs stored on the server that are user or server changeable, the challenge framework needs to be leveraged again in the following manner:

1. The authenticating agent issues an authentication request with username, organization name, and a passcode.
2. The server finds a match for the provided OTP; however, it determines that it is necessary for the user to change their PIN, therefore it issues a challenge to the client containing a PIN change string ("Your PIN has expired. Please enter a new PIN:") and a state attribute.
3. The authenticating agent responds to the challenge by returning the new PIN and the state attribute.

See the localization note in the section "Challenge-Response" page 9.

Static Password Authentication

SAS offers also the option of static password authentication, and enables the user to change their password. The challenge response architecture can be used to facilitate this functionality, in the following manner:

1. Authenticating agent issues an authentication request with username, organization name, and a static passcode.
2. If the user is not required to change the static password and it is correct, the server returns access-accept. If the user is required to change the static password, a challenge message will be issued to client ("Your password has expired. Please enter a new password:") and a state attribute.
3. The authenticating agent responds to the challenge by issuing an access request message with their username, organization, the new static password, and the state attribute.

See the localization note in the section "Challenge-Response" page 9.

Agent Key Files

The SAS API uses an encrypted key file to secure communication with the server. To accomplish this, a key file is loaded and registered with agents, and then a matching key is registered with the authentication server.

Logging

The Java Authentication API will log messages to the log file **C:/log/JCcryptoWrapper-{date}.log**. This location is configurable through the JavaAPIManager **Logging** tab. The logging level can be adjusted through this tab to warning, info, debug, error, or critical.

API Example

Refer to the sample code in **Main.java** for an example of how to call the **Authenticate** and **VerifySignature** methods.

To use the API example, the following are required:

- BSIDJavaAPI.jar (needed for Windows and Linux)
- NetBeans IDE or Eclipse, or any other Java development tool

Deployment Requirements and Deployment for Windows

To deploy your completed application to another computer, the following are required to support the SAS API:

- Your own **.jar** file and **JCryptoWrapper.ini** file should be in the same directory.
- **The following files are required:**

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bsidkey\ Agent.bsidkey

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\ini\ JCryptoWrapperWin.ini

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\jar\ BSIDJavaAPI.jar

  For 32-bit operating systems the following files are required:

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bin\x86\ CryptoCOM.dll

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bin\x86\ JCryptoWrapperWin.dll

  For 64-bit operating systems the following files are required:

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bin\x64\ CryptoCOM.dll

  Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bin\x64\ JCryptoWrapperWin.dll

Deployment Requirements and Deployment for Linux

To deploy your completed application to another computer, the following are required to support the SAS API:

- Your own .jar file and JCryptoWrapper.ini file should be in the same directory.
- Mono framework installation is required only if you wish to run the JavaAPIManager.
- **The following files are required:**
  - /usr/local/cryptocard/javaapi/bsidkey/Agent.bsidkey
  - /usr/local/cryptocard/javaapi/ini/JCryptoWrapper.ini
  - /usr/local/cryptocard/javaapi/jar/BSIDJavaAPI.jar

  For 32-bit operating systems the following files are required:
  - /usr/local/cryptocard/javaapi/bin/x86/libcrypto.so -> libcrypto.so.1.0.0
  - /usr/local/cryptocard/javaapi/bin/x86/libcrypto.so.1.0.0
  - /usr/local/cryptocard/javaapi/bin/x86/libJCryptoWrapper.so
  - /usr/local/cryptocard/javaapi/bin/x86/libLinuxSOAP.so
  - /usr/local/cryptocard/javaapi/bin/x86/libssl.so -> libssl.so.1.0.0
  - /usr/local/cryptocard/javaapi/bin/x86/libssl.so.1.0.0

For 64-bit operating systems the following files are required:

- /usr/local/cryptocard/javaapi/bin/x64/libcrypto.so -> libcrypto.so.1.0.0
- /usr/local/cryptocard/javaapi/bin/x64/libcrypto.so.1.0.0
- /usr/local/cryptocard/javaapi/bin/x64/libJCryptoWrapper.so
- /usr/local/cryptocard/javaapi/bin/x64/libLinuxSOAP.so
- /usr/local/cryptocard/javaapi/bin/x64/libssl.so -> libssl.so.1.0.0
- /usr/local/cryptocard/javaapi/bin/x64/libssl.so.1.0.0

Configuration

Configuration of the Java Authentication API is done through the JavaAPIManager.

For configuration on Windows the following files are required:

- Program Files\CRYPTOCard\BlackShield ID\JavaAPI\ JavaAPIManager.exe
- Program Files\CRYPTOCard\BlackShield ID\JavaAPI\Nini.dll

For configuration on Linux the following files are required:

- /usr/local/cryptocard/javaapi/JavaAPIManager.exe
- /usr/local/cryptocard/javaapi/Nini.dll.

JavaAPIManager.exe uses **Nini.dll** to write configuration settings to **JCryptoWrapper.ini** file on Linux or **JCryptoWrapperWin.ini** on Windows.

On the **Communications** tab, for **Primary Server (IP and Port)** field, enter the IP address of the SAS server. For the **Agent Encryption Key File** field, browse to the **Agent.bsidkey** file, located at **Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bsidkey\Agent.bsidkey** on Windows, and **/usr/local/cryptocard/javaapi/bsidkey/Agent.bsidkey** on Linux.

On the **Policy** tab, **Enable Agent** option should be selected. The **Send Remote Client IP Address to SAS Server** option toggles whether or not the API sends the client's IP address to SAS, overriding the SAS automatic detection of the IP address.

Support Contacts

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, contact your supplier or Gemalto Customer Support. Gemalto Customer Support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Gemalto and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact Method	Contact Information	
Address	Gemalto 4690 Millennium Drive Belcamp, Maryland 21017, USA	
Phone	US	1-800-545-6608
	International	1-410-931-7520
Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the Gemalto Knowledge Base.	