SafeNet Authentication Service

Java Authentication API for Linux Developer Guide

gemalto
security to be free

Document Number: 007-012469-001, Rev. J

Release Date: January 2018

Contents

Preface

This document describes the SafeNet Authentication Service Authentication API that enables SAS agents to support all functionality required to interact with SafeNet Authentication Service (SAS). A SAS agent is a third-party application with embedded plug-in code that passes the collected user names and one-time passwords (OTPs) to a SAS server for authentication.

The SAS Authentication API is represented by a single Java class CRYPTOCardAPI. This class is a singleton - no public constructor. Class instance can be acquired using **getInstance** public method.

Audience

This document is intended for personnel responsible for maintaining your organization's security infrastructure. It is assumed that the users of this document are proficient with security concepts.

All products manufactured and distributed by Gemalto are designed to be installed, operated, and maintained by personnel who have the knowledge, training, and qualifications required to safely perform the tasks assigned to them. The information, processes, and procedures contained in this document are intended for use by trained and qualified personnel only.

Support Contacts

If you encounter a problem while installing, registering, or operating this product, refer to 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.

Customer Support Portal

The Customer Support Portal, at https://supportportal.gemalto.com, is a where you can find solutions for most common problems. The Customer Support Portal is a comprehensive, fully searchable database of support resources, including software and firmware downloads, release notes listing known problems and workarounds, a knowledge base, FAQs, product documentation, technical notes, and more. You can also use the portal to create and manage support cases.

NOTE: You require an account to access the Customer Support Portal. To create a new account, go to the portal and click on the **REGISTER** link.

Telephone Support

If you have an urgent problem, or cannot access the Customer Support Portal, you can contact Customer Support by telephone. Calls to Customer Support are handled on a priority basis.

Region	**Telephone number** (Subject to change. An up-to-date list is maintained on the Customer Support Portal)
Global	+1-410-931-7520
Australia	1800.020.183
China	North: 10800-713-1971 South: 10800-1301-932
France	0800-912-857
Germany	0800-181-6374
India	000.800.100.4290
Israel	180-931-5798
Italy	800-786-421
Japan	0066 3382 1699
Korea	+82 2 3429 1055
Netherlands	0800.022.2996
New Zealand	0800.440.359
Portugal	800.863.499
Singapore	800.1302.029
Spain	900.938.717
Sweden	020.791.028
Switzerland	0800.564.849
United Kingdom	0800.056.3158
United States	(800) 545-6608

1 Public Methods/Functions

public static CRYPTOCardAPI getInstance()

This method returns a singleton instance of the CRYPTOCardAPI class. This method also does some internal initializations and loads a native interface (JNI class) in a custom class loader. On failure to load the native class in a custom class loader, this function may send this error message to std error out:

System.out.println("Failed to load class in custom class loader");

Loading in a custom class loader provides ability to load multiple native libraries instances - each per an application.

public synchronized void 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 the INI file is not persent at the default location, the current execution path will be explored.

public synchronized void LoadJNILibrary() throws UnsatisfiedLinkError, Exception

This function loads the SafeNet encryption and logging native library. On Linux-like systems, this function tries to load the native library (libJCrtptoWrapper.so) from the path exported using **LD_LIBRARY_PATH**.

If LD_LIBRARY_PATH is not defined in application/server launch scripts, then the above-mentioned library must be in the Linux normal libraries folders (for example, /lib, /etc/lib or /etc/lib64). Certain J2EE servers have their own paths defined by the **–Djava.library.path** variable, which can also be used.

On Linux-like systems, to make sure there are no unresolved dependencies for **libJCryptoWrapper.so**, run ldd **./libJCryptoWrapper.so** to confirm that all the dependencies can be resolved.

On Windows, it will try to find **JCryptoErapper.dll** in the system **PATH** environment variable. On Windows systems, the management tool adds this path to the system`s **PATH** environment variable, making it visible to Java applications. But if it is desired to load binaries from a different location (for example, multiple applications using their own binaries), this function should NOT be used.

If this function succeeds, the native library loads the INI file and initializes logging and encryption functions. The log file gets created at the location defined in the INI file.

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

If this function fails to load the native library, it will throw an **UnsatisfiedLinkError** or **Exception** to std out. If the INI loading function or the encryption/decryption initialization fails, it will throw this exception:

```
System.out.println(String.format("SafeNet-> CRYPTOCardAPI -> LoadJNILibrary ->
LD_LIBRARY_PATH = %s", System.getenv("LD_LIBRARY_PATH")));

throw new Exception("SafeNet-> CRYPTOCardAPI -> LoadJNILibrary -> Initialize FALIED.\n"
       + "Please check LD_LIBRARY_PATH on Linux.\n"
       + "Please run ldd libJCryptoWrapper.so to find missing dependencies.\n"
       + "On Windows Systems, please check Path Env. variable and check missing dependencies
using dependency walker.");
```

Read permission must be set on defined paths before attempting to load any binaries. Write permissions must be set on the directory where the log file has to be generated.

public synchronized void LoadJNILibrary(String LoadFromLocation) throws UnsatisfiedLinkError, Exception

This function loads the SafeNet encryption and logging native library (for Linux-like systems, it's **libJCryptoWrapper.so** and for Windows, it's **JCryptoWrapper.dll)** from the desired location. This function should be used if multiple applications will be using the API. In this case, if the API throws an **UnsatisfiedLinkError** exception with the message "class already loaded," you can copy and rename the same binary and load it instead. For example, you can copy and rename **libJCryptoWrapper.so** as **libJCryptoWrapper_Ex.so,** and then load both using different names in different applications. In the example above, one application will load **libJCryptoWrapper.so,** and another application will load **libJCryptoWrapper_EX.so**. You can also use **setINIPath** to define different INI files for different applications.

If this function fails to load the native library, it will throw an **UnsatisfiedLinkError** or **Exception** to std out. If the INI loading function or the encryption/decryption initialization fails, it will throw this exception:

```
System.out.println(String.format("SafeNet-> CRYPTOCardAPI -> LoadJNILibrary ->
LoadJNILibrary Path = %s", LoadFromLocation));

throw new Exception("SafeNet-> CRYPTOCardAPI -> LoadJNILibrary -> Initialize FALIED.\n"
       + "Please check LD_LIBRARY_PATH on Linux.\n"
       + "Please run ldd libJCryptoWrapper.so to find missing dependencies.\n"
       + "On Windows Systems, please check Path Env. variable and check missing dependencies
using depedency walker.");
```

Read permission must be set on defined paths before attempting to load any binaries. Write permissions must be set on the directory where the log file has to be generated.

public void Authenticate(String[] arrData)

This function provides SAS authentication and challenge generation functionality. It must be called only after successful loading and initialization.

String[] arrData details:

arrData[0]	UserName	In Value	
arrData[1]	Organization	In Value	Optional. Normally blank, except in special cases.
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. Optional. Normally blank, except in some special cases.

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.
  - [OTP+PIN] – If it’s configured this way in SAS.
  - [StaticPassword] – User has a static password enabled or is responding to a static password change.
  - Finally, this parameter may also be set to **empty string or to a single character to indicate a challenge is required. If a challenge generation is required, please send empty state in arrData[4] as well.**
- 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 response to a Challenge, the same state should be passed back in state element i.e. arrData[4], which was returned by the challenge generation call.
- ChallengeData – Data returned as the response to the challenge.
- ChallengeMessage – Returned user message appended with 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 – SafeNet Authentication Service (SAS) 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. Under normal circumstances, this should be left empty.

public void VerifySignature(String[] arrData)

This function verifies the token's signature for a given hash.

String[] arrData details:

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 parameters should be avoided. Instead, 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.

public BufferedImage getGridSureGrid(String BSIDChallenge) throws Exception

This function creates and returns a GrIDsure grid from the received SAS challenge.

getGridSureGrid method returns instance of BufferedImage class i.e. bitmap.

Redundant Function – Not required to be used: public BufferedImage getGridSureLogo() throws Exception

getGridSureLogo method returns the GrIDsure logo (instance of BufferedImage class) from a String saved in this function.

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

public String getSoapPayload(String[] arrData) throws Exception

This function gives the API user ability to use their own SOAP transport layer. The API itself use following java packages for SOAP calls:

```
java.net.Authenticator;
java.net.HttpURLConnection;
java.net.InetSocketAddress;
java.net.MalformedURLException;
java.net.Proxy;
java.net.Socket;
java.net.SocketAddress;
java.net.URL;
```

API fully handles HTTP, HTTPS and HTTP/HTTPS via HTTP Proxy with basic authentication. But, for any reason if it is not desired to use built in functionality, one may use this function to get the input parameter required to make SOAP based authentication call.

Call to this procedure will return encrypted SOAP payload accepted by SafeNet Authentication Web Service (SAS). This payload should be enclosed inside a SOAP envelop to be sent to SAS server.

Example SOAP Request With Headers and Soap Envelop:

The following is a sample SOAP 1.2 request and response. The place holders shown need to be replaced with actual values.

```
POST /TokenValidator/TokenValidator.asmx HTTP/1.1

Host: 192.168.40.124

Content-Type: application/soap+xml; charset=utf-8 Content-Length: length


<?xml version="1.0" encoding="utf-8"?>

<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">

<soap12:Body>

<Authenticate xmlns="http://www.cryptocard.com/blackshield/">

<CRYPTOCardData>ENCRYPTED PAYLOAD HERE - RETURNED BY THIS
FUNCTION</CRYPTOCardData>

</Authenticate>

</soap12:Body>

</soap12:Envelope>
```

If the SOAP call is successful, you should get this response:

```
HTTP/1.1 200 OK

Content-Type: application/soap+xml; charset=utf-8

Content-Length: length


<?xml version="1.0" encoding="utf-8"?>

<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">

<soap12:Body>

<AuthenticateResponse xmlns="http://www.cryptocard.com/blackshield/">

<AuthenticateResult>ENCRYPTED AUTHENTICATION RESULT</AuthenticateResult>

</AuthenticateResponse>

</soap12:Body>

</soap12:Envelope>
```

To get result back, pass this "**ENCRYPTED AUTHENTICATION RESULT**" to the getResultFromAuthenticateResult function.

String[] arrData Details:

arrData[0]	UserName	In Value	
arrData[1]	Organization	In Value	Optional. Normally blank, except in special cases.
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	Not Applicable	
arrData[9]	ErrorMessage	Return Value -> Error Message	
arrData[10]	InIPAddress	In Value	

Function throws an exception if input is invalid - Array Len is less than 11, No User Name or it fails to encrypt.

All input and output values must follow the details provided in the Authenticate function.

public String[] getResultFromAuthenticateResult(String encryptedAuthenticateResult) throws Exception

Call this after you have an encrypted result from a SOAP call to SafeNet Authentication Web Service.

This function returns the following array:

Challenge	= 0	Return Value (If returned by Authentication Service)
State	= 1	Return Value (If returned by Authentication Service)
ChallengeData	= 2	Return Value (If returned by Authentication Service)
ChallengeMessage	= 3	Return Value
ReturndResult	= 4	Return Value
ErrorMessage	= 5	Return Value -> Error Message for Logging or client, if any

Example SOAP Request With Headers and Soap Envelop:

If SOAP call is successful, you should get this response:

HTTP/1.1 200 OK

Content-Type: application/soap+xml; charset=utf-8

Content-Length: length

```
<?xml version="1.0" encoding="utf-8"?>

 <soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">

<soap12:Body>

      <AuthenticateResponse xmlns="http://www.cryptocard.com/blackshield/">

      <AuthenticateResult>ENCRYPTED AUTHENTICATION RESULT</AuthenticateResult>

      </AuthenticateResponse>

</soap12:Body>

</soap12:Envelope>
```

encryptedAuthenticateResult is value of **AuthenticateResult** returned by SAS Web Service.

This function throws exception if input is null, empty or it fails to decrypt

2 Use Case Scenarios

SAS supports token-side and server-side PINs used in either QuickLog or Challenge/Response mode. Agents must support challenges, inner/outer window authentication, and static password authentication.

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 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 authentication request with username, organization name, and a blank passcode.
2. Server responds with a challenge message containing a challenge string i.e.: “challenge: 19863257”, and a state attribute.
3. Authenticating agent responds to the challenge by issuing another authentication request with their username, organization name, a response, and the state attribute.

NOTE: 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 used also 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 follow sequence illustrates this functionality:

1. Authenticating agent issues authentication request with username, organization name, and a passcode.
2. The server finds a match for the provided OTP in the outer window, therefore it 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. Authenticating agent responds to the challenge by issuing another authentication request with their username, organization name, a response, and the state attribute.

Refer to the localization note in the “Challenge-Response” section on page 14.

PIN Styles

The following PIN styles are supported in SAS:

- 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.

Stored on server, user or server changeable

In the case of PINs that are stored on the server, yet are user or server changeable, the challenge framework needs to be leveraged again, as follows:

1. Authenticating agent issues authentication request with username, organization name, and a passcode.
2. The server finds a match for the provided OTP. However, it determines that the user must 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.

Refer to the localization note in the “Challenge-Response” section on page 13.

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.

Refer to the localization note in the "Challenge-Response" section on page 13.

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 to warning, info, debug, error, or critical.

3 Additional Information

API Example

See 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 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 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/libJCryptoWrapper.so
- **For 64-bit operating systems the following files are required:**
  - /usr/local/cryptocard/javaapi/bin/x64/libJCryptoWrapper.so

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:

- **Primary Server (IP and Port):** Enter the IP address of the SAS server.
- **Agent Encryption Key File:** Browse to and select the **Agent.bsidkey** file.

The file **Agent.bsidkey** is located at **\Program Files\CRYPTOCard\BlackShield ID\JavaAPI\bsidkey\Agent.bsidkey** on Windows or **/usr/local/cryptocard/javaapi/bsidkey/Agent.bsidkey** on Linux.

On the **Policy** tab:

- **Enable Agent:** Do not select this check box.
- **Send Remote Client IP Address to SAS Server:** This option determines whether or not the API should sends the client's IP address to SAS, overriding SAS's automatic detection of the IP address.

WLS SSL Errors if SSL Connection to SAS is Used

The certificate policy is different inside WebLogic than in a stand-alone Java program. This page advises to use a Sun implementation instead of WebLogic:

http://webtech-kapil.blogspot.com/2010/06/javalangclasscastexception.html

They advise to start WebLogic with the following flag:

```
-DUseSunHttpHandler=true
```

This will use standard Sun's implementation of SSL.

When SOAP transport in the INI file is via Java code, on WLS you might get SSL connection, certificate validation, and SSL handshake error. All these errors can be prevented by adding this to your WLS startup script at:

<WLS INSTALL PATH>/oracle/Middleware/user_projects/domains/OAMdomain/bin/setDomainEnv.sh

Just below the Comments block:

```
# ***************************************************************************
# This script is used to setup the needed environment to be able to start Weblogic Server in this domain.
#
# This script initializes the following variables before calling commEnv to set other variables:
 #
# WL_HOME         - The BEA home directory of your WebLogic installation.
# JAVA_VM         - The desired Java VM to use. You can set this environment variable before calling #
                  this script to switch between Sun or BEA or just have the default be set.
 # JAVA_HOME       - Location of the version of Java used to start WebLogic
#                   Server. Depends directly on which JAVA_VM value is set by default or by the environment.
# USER_MEM_ARGS   - The variable to override the standard memory arguments
#                   passed to java.
 # PRODUCTION_MODE - The variable that determines whether Weblogic Server is started in production mode. # DOMAIN_PRODUCTION_MODE
#                 - The variable that determines whether the workshop related settings like the debugger,
#                   testconsole or iterativedev should be enabled. ONLY settable using the
#                   command-line parameter named production
#                   NOTE: Specifying the production command-line param will force
#                         the server to start in production mode. #  # Other variables used in this script include:
# SERVER_NAME     - Name of the weblogic server.
# JAVA_OPTIONS    - Java command-line options for running the server. (These
#                   will be tagged on to the end of the JAVA_VM and
#                   MEM_ARGS)
#
# For additional information, refer to "Managing Server Startup and Shutdown for Oracle WebLogic Server" # (http://download.oracle.com/docs/cd/E17904_01/web.1111/e13708/overview.htm).
# ***************************************************************************
```

#added to avoid SSL errors

JAVA_OPTIONS="-DUseSunHttpHandler=true"

You must configure, at a minimum, the following INI keys (Linux):

```
PrimaryProtocol=https

PrimaryServer=Primary SAS Server Host OR IP

PrimaryServerPort=443


SecondaryProtocol=https

SecondaryServer=Secondary SAS Server Host OR IP

SecondaryServerPort=443


LogFile=/usr/local/cryptocard/javaapi/log/javaapi-{date}.log

LogLevel=3


; If set to 0, SAS Server certificate checks will be forced while communication
with SAS Servers.

; It is recommended to force certificate checks if SAS cloud Service is in use.

; For inhouse SAS deployments, with self-signed certificate, please set this
setting to 1

; If you change this setting to 0, SAS server must have a valid certificate

; Valid only for HTTPS protocol

IGNORE_CERTIFICATE_ERRORS=0
```

NOTE:. For Windows, use the Management UI to configure the INI file. You can configure it manually as well. On Linux systems, you may use the same UI if mono framework is installed.

HTTP Proxy (Windows and Linux)

If your organization uses a proxy server to access an extranet or intranet, configure proxy settings in the INI file as well. The agent software can work with an HTTP proxy only with basic or anonymous authentication.

```
;User Optional, No user, no password. No checks on IP/Host and Port so put correct. No HTTPS proxy

USE_PROXY=0

PROXY_SERVER=127.0.0.1

PROXY_PORT=8080

PROXY_USER=User

PROXY_PASSWORD=Password
```

Self-Signed Certificates (In-house SAS)

IGNORE_CERTIFICATE_ERRORS must be set to 1 to avoid SSL errors if self-signed certificates are used (in-house SAS deployments). This key is valid for both Windows and Linux.