



SafeNet ProtectApp for .NET User Guide

Software Version: 5.1.1
Documentation Version: 20100913

Preface

© 2010 SafeNet, Inc All rights reserved

All intellectual property is protected by copyright. All trademarks and product names used or referred to are the copyright of their respective owners. No part of this document may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, chemical, photocopy, recording or otherwise without the prior written permission of SafeNet.

SafeNet 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, SafeNet reserves the right to revise this publication and to make changes from time to time in the content hereof without the obligation upon SafeNet to notify any person or organization of any such revisions or changes.

SafeNet invites constructive comments on the contents of this document. These comments, together with your personal and/or company details, should be sent to the address below.

4690 Millennium Drive
Belcamp, Maryland 21017
USA

Disclaimers

The foregoing integration was performed and tested only with specific versions of equipment and software and only in the configuration indicated. If your setup matches exactly, you should expect no trouble, and Customer Support can assist with any missteps. If your setup differs, then the foregoing is merely a template and you will need to adjust the instructions to fit your situation. Customer Support will attempt to assist, but cannot guarantee success in setups that we have not tested.

This product contains software that is subject to various public licenses. The source code form of such software and all derivative forms thereof can be copied from the following website: <http://c3.safenet-inc.com/>

We have attempted to make these documents complete, accurate, and useful, but we cannot guarantee them to be 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.

Technical Support

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, please contact your supplier or SafeNet support.

SafeNet 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 SafeNet and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Technical Support Contact Information:

Phone: 800-545-6608, 410-931-7520
Email: support@safenet-inc.com

Table of Contents

ABOUT THIS GUIDE.	8
Using This Guide	8
Documentation Conventions	9
Code Samples	9
Notes and Cautions	9
CHAPTER 1 OVERVIEW	10
General System Architecture	10
Hardware and Software Requirements	11
Supported Cryptographic Operations	12
Supported Content	12
CHAPTER 2 INSTALLING PROTECTAPP FOR .NET	13
Obtaining ProtectApp for .NET Software	13
Installing ProtectApp for .NET	14
The NAE_Properties_Config Registry Key	16
The Installed Directory	17
Upgrading ProtectApp for .NET	17
Properties File	18
Examples Folder	18
Repairing ProtectApp for .NET	18
Uninstalling ProtectApp for .NET	18
The Sample Application	19
Compiling the Sample Application with Visual Studio 2010	19
Before You Begin	20
CHAPTER 3 CONFIGURING THE PROPERTIES FILE	21
Editing the Properties File	21
Renaming the Properties File	22
The Parameters	22
Version	23
NAE_IP	23
Port	23

Protocol	23
Use_Persistent_Connection	23
Size_of_Connection_Pool	24
Connection_Timeout	24
Connection_Idle_Timeout	24
Connection_Retry_Interval	25
Cluster_Synchronization_Delay	25
EdgeSecure_Name	25
Cipherspec	26
CA_File	26
Cert_File	26
Key_File	27
Passphrase	27
Symmetric_Key_Cache_Enabled	27
Symmetric_Key_Cache_Expiry	28
Persistent_Cache_Enabled	28
Persistent_Cache_Directory	28
Persistent_Cache_Expiry_Keys	29
Persistent_Cache_Max_Size	29
Log_Level	29
Log_File	29
Log_Rotation	30
Log_Size_Limit	30
Reading System Properties From the Windows Registry	30
Setting Properties in the Registry via the Sample Configuration	31
Manually Setting Properties in the Registry	32
CHAPTER 4 CONNECTING TO A SERVER	33
Overview	33
How it Works	33
Related IngrianNAE.properties Parameters	34
CHAPTER 5 CONNECTION POOLING	36
Connection Pools	36
How it Works	36
Related IngrianNAE.properties Parameters	37
Examples	38

CHAPTER 6	LOAD BALANCING GROUPS	39
	Overview	39
	How it Works	40
	Related IngrianNAE.properties Parameters	41
	Examples	42
CHAPTER 7	MULTI-TIER LOAD BALANCING	43
	Overview	43
	How it Works	44
	Related IngrianNAE.properties Parameters	45
	Examples	47
CHAPTER 8	SETTING UP SSL	48
	SSL Overview	48
	SSL Configuration Procedures	49
	Creating a Local CA	50
	Creating a Server Certificate Request on the Management Console	50
	Signing a Server Certificate Request with a Local CA	50
	Importing a Server Certificate to the DataSecure Appliance	51
	Downloading the Local CA Certificate	52
	SSL Walkthrough for SafeNet Clients	52
	SSL with Client Certificate Authentication Overview	56
	SSL with Client Certificate Authentication Procedures	57
	Generating a Client Certificate Request with req.exe	58
	Signing a Certificate Request and Downloading the Certificate	59
	Installing a CA Certificate on the Server	60
	Adding a CA to a Trusted CA List Profile	60
	SSL with Client Certificate Authentication Walkthrough for DataSecure Clients	60
CHAPTER 9	CREATING AN NAESESSION	64
	Creating a Global Session to a DataSecure	64
	Creating an Authenticated Session to a DataSecure	64
CHAPTER 10	WORKING WITH KEYS	65
	Listing All Keys Available on the DataSecure	65
	Obtaining an Instance of a Key	65
	Obtaining an Instance of a Key - Alternate Method	66
	Deleting a Key Using the Key Name	66
	Creating a Key	66

Importing a Symmetric Key	67
Importing an Asymmetric Key	68
Setting the Key Mode and Padding	68
CHAPTER 11 USING VERSIONED KEYS	69
Overview	69
Creating a Versioned Key	70
Creating a New Version	70
Activate, Restrict, or Retire a Version	70
Using a Versioned Key to Encrypt, Sign, and MAC	70
Using a Versioned Key to Decrypt, SignV, and MACV	71
CHAPTER 12 SYMMETRIC KEY CACHING	72
Overview	72
Supported Functions	73
How it Works	73
Related IngrianNAE.properties Parameters	73
Logging	74
CHAPTER 13 PERSISTENT KEY CACHING	75
Overview	75
Supported Functions	76
How it Works	76
Related IngrianNAE.properties Parameters	77
Logging	78
Tips	78
Pre-Loading Keys	78
Troubleshooting	79
CHAPTER 14 WORKING WITH CERTIFICATES	80
Importing a Certificate	80
Exporting a Certificate	81
Exporting a CA Chain	82
Deleting a Certificate	82
CHAPTER 15 ENCRYPTING AND DECRYPTING DATA	83
Encrypting a String Using an AES Key	83
Decrypting a String Using an AES Key	85
Encrypting a String Using an RSA Key	86

- Decrypting a String Using an RSA Key 86
- Encrypting a File 87
- Decrypting a File 89
- CHAPTER 16 GENERATING A MAC 91
 - Creating a MAC 91
- CHAPTER 17 USING PROTECTAPP FOR .NET API 92
 - Enabling Users to Perform Administrative Operations 92
 - Overview 92
 - Thread Safety 93
 - Exceptions 93
 - Supported Functions 93
 - Supporting Calls 93
 - Connection Calls 94
 - Key-related APIs 96
 - MAC/Hash-related APIs 103
- INDEX 106

About This Guide

This introductory chapter gives a brief summary of the book's contents, identifies the audience, explains how to best use the written material, discusses the documentation conventions used, and provides instructions for contacting technical support.

This chapter contains the following sections:

Using This Guide **8**

Documentation Conventions **9**

Using This Guide

Generally speaking, our user guides are written for network administrators, security engineers, database administrators, application developers, and other technology professionals responsible for daily operations in support of data security. The written material we provide describes how to configure and operate our products and assumes a working knowledge of networking, computer security, database management, and cryptography.

This specific book is designed for advanced developers familiar with the .NET framework, and the DataSecure appliance.

Documentation Conventions

This section describes the formatting conventions used in this manual to explain code samples, special notes and cautions.

Code Samples

Samples code is illustrated in the format shown below. Much of this guide includes pieces of sample code that you will not be able to compile by itself.

```
NAESession * session = new NAESession();
```

Notes and Cautions

The following paragraph formats are used to highlight information in the text:

Note: A note conveys information that supplements the preceding text. This information may refer to certain situations or a specific technical setup.

Important! An important note is a very significant piece of information required for the completion of a task.

Tip: A tip helps you apply the information in the preceding text.

WARNING! A warning advises you to exercise care when working around specified equipment conditions. Heeding a warning can prevent personal injury, system disruption, or equipment damage.

Overview

This document describes how to integrate the ProtectApp for .NET with your back-end application servers and it gives code samples that illustrate how you might customize the ProtectApp for .NET as your application requires. This chapter provides a description of the high-level architecture of the DataSecure Platform, lists the hardware and software requirements, and discusses the supported cryptographic operations.

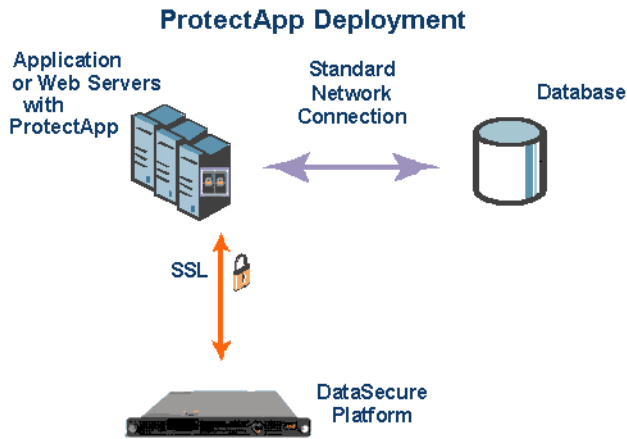
This chapter contains the following sections:

General System Architecture	10
Hardware and Software Requirements	11
Supported Cryptographic Operations	12
Supported Content	12

General System Architecture

The DataSecure Platform consists of two required components – the client (the ProtectApp for .NET in this case) and the DataSecure – and, in some cases, an optional Database Connector.

The diagram below shows a high-level network diagram of a typical deployment of the DataSecure Platform. Whenever necessary, the DataSecure client (application, web, database servers) makes requests via one of the ProtectApp Clients or the XML interface for cryptographic operations to be performed by the DataSecure Appliance. The DataSecure Appliance performs all desired cryptographic operations and returns data to the application that made the request. At that point, if the client is an application, it might want to store the data in a database, or it might want to return the data to a client over the internet. This unique method of providing cryptographic functionality over the network creates an extremely simple, scalable, and secure solution to backend data encryption, integrity checking and fingerprinting (hashing). An example configuration is illustrated below.



The ProtectApp for .NET is installed on all the back-end servers that might be making requests for cryptographic operations. All applications, servlets, or scripts see a conventional .NET interface and issue simple commands to the DataSecure to perform cryptographic operations. Instead of bogging down back-end server applications with cryptographic operations, the DataSecure performs all such operations.

Hardware and Software Requirements

The hardware and software required to deploy the DataSecure Platform are listed below.

Required Equipment

- **DataSecure appliance:** This is available in various hardware configurations and comes standard with two 10/100 Ethernet interfaces for connecting to the back-end servers. Options are available for redundant power supplies, redundant fans, and copper and fiber Gigabit Ethernet versions.
- **ProtectApp for .NET:** This is provided in the form of a DLL file.

Other Requirements

- Windows platforms with .NET Framework Version 2.0 or above

Supported Cryptographic Operations

The ProtectApp for .NET exposes functionality to allow the user to implement data privacy, confidentiality and integrity in a simple, scalable and secure manner. The operations supported are listed below.

Security Provided	Algorithm	Functions Supported
Data Privacy and Confidentiality (Symmetric)	AES, DESede, DES	• Encrypt / Decrypt
Data Privacy and Confidentiality (Asymmetric)	RSA	• Encrypt / Decrypt
Data Integrity	HmacSHA1, HmacSHA2	• MAC / MAC Verify

Supported Content

There are no restrictions on the type of data and content that the DataSecure can secure. Whether it is a 10 byte string of data, a 10K image, a 1 MB text file, a 10MB PDF file, a financial spreadsheet, or a line of code, the NAE Server can perform all desired cryptographic operations. In short, the DataSecure Platform can handle any type of data or content.

Installing ProtectApp for .NET

This chapter describes how to obtain, install, upgrade, and uninstall ProtectApp for .NET.

This chapter contains the following sections:

Obtaining ProtectApp for .NET Software	13
Installing ProtectApp for .NET	14
Upgrading ProtectApp for .NET	17
Repairing ProtectApp for .NET	18
Uninstalling ProtectApp for .NET	18
The Sample Application	19

Obtaining ProtectApp for .NET Software

You can obtain the appropriate installation program by logging into the Customer Support web site. All installation programs adhere to the following naming convention:

`partnumber_software_platform_version.format`

The following table explains the naming convention.

Value	Description
partnumber	The part number or the unique identifier of the software.
software	The software name.
platform	The name of the platform, such as Microsoft Windows 32 bit.
version	The version number of the software.
build	The specific build in the release for which the installer was created.
format	The format in which the installation program is delivered; usually, an executable (EXE) file.

For example,

`611-009849-001_datasecure_protectapp_dotnet_32bit_windows_v5.1.1.000-010.exe`

Installing ProtectApp for .NET

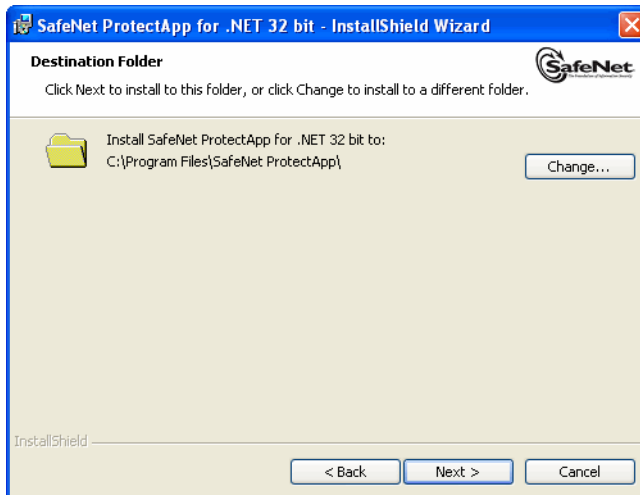
Important! Before installing ProtectApp for .NET, you must have the .NET Framework Version 2.0 or above installed. The installer will detect if you do not have the framework and will prompt you to install it.

To install ProtectApp for .NET, follow the steps below.

- 1 Double-click `setup.exe`. This launches an InstallShield Wizard that walks you through the installation process. Click **Next**.

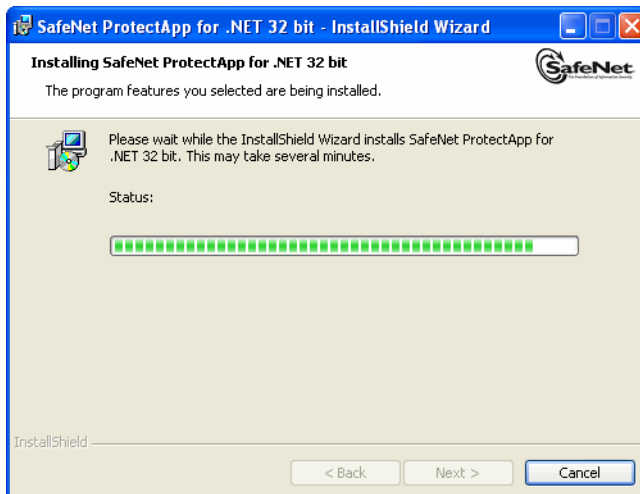


- 2 Provide a destination for the program files and click **Next**.

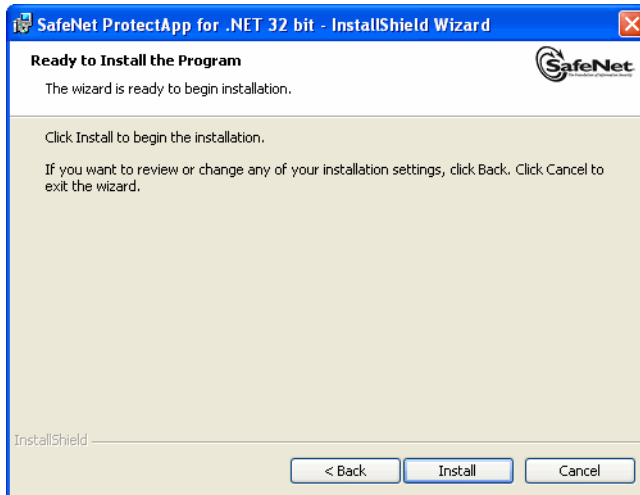


The directory you specify will contain all of the files described in “The Installed Directory” on page 17.

- 3 Click **Install** to begin the installation.



- 4 The installer will display the progress of the installation.



- 5 Once the necessary files are installed, click the **Finish** button to exit the InstallShield Wizard.



The NAE_Properties_Config Registry Key

The installation program creates a registry key called ConfigFilename with the following value:

- **ConfigFilename** - specifies the location of the IngrianNAE.properties file. There is no default value for this string; rather, the value of the string is set when you specify where to install ProtectApp for .NET. You should note that this string specifies a path *and* a file name.

The `NAE_Properties_Config` key is located in `My Computer \ HKEY_LOCAL_MACHINE \ SOFTWARE \ Ingrian`.

The Installed Directory

The installation program creates the following directory structure, which includes a sample application written in Visual Basic, Visual C++, and Visual C#.

The `examples` folder contains sample code and applications written in Visual Basic, Visual C++, and Visual C#. The sample application is detailed in “The Sample Application” on page 19.

The `ingdnp.dll` file is the dynamic-link library that contains the ProtectApp for .NET API.

The `IngrianNAE.properties` file contains the system properties. For more information, see “IngrianNAE.properties File Overview”.

The files `req` and `openssl.conf` are used to generate certificate requests. The `req` file needs to be in the same directory as `openssl.conf` to run. Likewise, you must call `req` from within the directory where it resides. You might need to generate client certificates for your client applications if you are requiring client certificate authentication. This topic is discussed further in “SSL with Client Certificate Authentication Overview” on page 56.

```
\SafeNet ProtectApp
  \DotNet
    \examples
      \VB
      \VC
      \VC#
      ingdnp.dll
      IngrianNAE.properties
      openssl.conf
      req.exe
```

The file `SampleRegistryConfig.reg` is a sample registry file. For more information on the sample registry file, see “Reading System Properties From the Windows Registry” on page 30.

Upgrading ProtectApp for .NET

Important! Before upgrading ProtectApp for .NET, close all associated applications. Examples of such applications include Windows Explorer, Command Prompt, and Visual Studio etc.

To upgrade ProtectApp for .NET, follow the installation procedure described in “Installing ProtectApp for .NET” on page 14. The installation will detect if you are upgrading and alter the process accordingly.

The upgrade process overwrites all of your existing files except the properties file and the `examples` folder.

Properties File

The process creates a backup copy of your existing configuration file (for example, 5.0.0 IngrianNAE.properties) in the `backup` folder. It does not add any new parameters to the existing 5.0.0 IngrianNAE.properties file.

A new properties file, prefixed with the software version (for example, 5.1.1.000.013.IngrianNAE.properties) is created for the new version. Now, you can use the 5.0.0 IngrianNAE.properties file to copy the existing configuration to the new properties file. In case you accidentally make any changes to the 5.0.0 IngrianNAE.properties file, you can revert the changes by referring to the properties file stored at the `backup` folder.

You must manually add any new parameters to the new properties file to use new features.

This guide provides a complete list of parameters in Chapter 3, “Configuring the Properties File”. Compare that list with your properties file, and insert the new parameters in the order shown.

Examples Folder

A new folder named `<VERSION>.examples` is created, where `<VERSION>` is the version number of the new installation. For example, if the new version number is 5.1.1.000.013, a new examples folder, `5.1.1.000.013.examples`, gets created. The `examples` folder of previous installation remains intact.

Repairing ProtectApp for .NET

The procedure to repair ProtectApp for .NET installation is same as “Upgrading ProtectApp for .NET” on page 17.

Uninstalling ProtectApp for .NET

To uninstall ProtectApp for .NET:

- 1 In your Microsoft Windows Operating System, navigate to Start > Settings > Control Panel > Add or Remove Programs.
- 2 Select the SafeNet ProtectApp for .NET32 bit / SafeNet ProtectApp for .NET 64 bit and click **Remove**.

The Sample Application

The ProtectApp for .NET is distributed with source code written in Visual Basic, Visual C++, and Visual C#. Once compiled, this source code can be run to test your installation.

Important! Compile this code using MS Visual Studio 2005, Visual Studio 2008, and Visual Studio 2010. The compilation of this code is not supported on Visual Studio 2003.

The sample application prompts you for a valid user name and password, and uses that information to log in to the DataSecure specified in the `IngrianNAE.properties` file. The program then searches for a key named `ingrian_example_key`. If one doesn't exist, it creates a deletable 128-bit AES key. (On Non-FIPS servers, this key is exportable.)

You can enter up to 127 bytes of clear text. The program will display the encrypted and decrypted values, and then exit.

Compiling the Sample Application with Visual Studio 2010

To compile the sample application with Microsoft Visual Studio 2010, you first need to convert the sample application project files into the Visual Studio 2010 project files.

Converting Sample Application Project Files into Visual Studio 2010 Project Files

You can convert the supplied project files into Microsoft Visual Studio 2010 project files by using Visual Studio 2010 Translator.

While converting the Visual C++ project, the x64 tools should be installed with Visual Studio 2010. If the x64 tools are not installed, remove the configuration for x64 tools from the project file.

Using Sample Applications with .NET Framework 4.0

The supplied sample application can be targeted for the .Net Framework 4.0 by including the `app.config` file in the corresponding project.

Content of a sample `app.config` file is given below:

```
<?xml version="1.0"?>
<configuration>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

For Visual C++, the app.config file needs to be copied manually to the Debug or Release folder. The app.config should be named as <exename>.exe.config, where <exename> is the name of the executable file.

Before You Begin

Before you can run the sample application, you must create a user on a DataSecure (if you have not done so already) and modify the properties file so that the configuration values for IP address, port and protocol correspond to the configuration of your DataSecure. Follow the steps below to modify the properties file:

- 1 Navigate to the directory into which you installed ProtectApp for .NET. The default location is:

```
C:\Program Files\SafeNet ProtectApp\
```

- 2 Open `IngrianNAE.properties` in a text editor such as Notepad.

- 3 Modify the following parameters:

- **NAE_IP** – IP address of the DataSecure that you will be connecting to. We recommend that you specify only one DataSecure in this initial testing period.
- **NAE_Port** – Port that the DataSecure is listening on.
- **Protocol** – Make sure this parameter is set to TCP.
- **Log_File** – Name for the log file. You should specify a name *and* a path. If you do not specify a value for the log file, a file called Logfile will be created in the same directory as the executable used to test the installation. The user running your client application must have permission to write to the log file, and to create new files in the directory where the log file(s) will be created.

Note: The IP address, port, and protocol parameters you specify must correspond to the values specified on the DataSecure.

- 4 Save your changes and close `IngrianNAE.properties`.

Now you can test your installation.

Configuring the Properties File

This chapter lists the contents of the IngrianNAE.properties file. The properties file defines, among other things, the IP address, port, and protocol of the DataSecures to which your client connects.

This chapter contains the following sections:

- Editing the Properties File **21**
- Renaming the Properties File **22**
- The Parameters **22**
- Reading System Properties From the Windows Registry **30**

Editing the Properties File

The values in the properties file are case-sensitive. yes is not YES. tcp is not TCP. Follow the example of the default properties file.

You can comment-out values using #. You'll see that the properties file is delivered with both **EdgeSecure_Name** and **Cipher_Spec** commented-out. You may want to use comments to save settings when troubleshooting. For example, you could store commonly used NAE_IP addresses like this:

```
NAE_IP=10.0.0.2  
#NAE_IP=10.0.0.3  
#NAE_IP=10.0.0.4
```

When editing parameters that use time values, you can use the following abbreviations:

- ms - milliseconds. e.g. 4500ms for 4.5 seconds.
- s - seconds. e.g. 30s for 30 seconds.
- m- minutes. e.g. 5m for 5 minutes.
- h - hours. e.g. 10h for 10 hours.
- d - days. e.g. 2d for 2 days.

If you do not include an abbreviation, the default time unit is used. For most time-related values the default is milliseconds. For **Symmetric_Key_Cache_Expiry** and **Persistent_Cache_Expiry_Keys**, the default is seconds.

Renaming the Properties File

Although the file is named `IngrianNAE.properties`, you can rename the file to any valid name. If you do rename the file, you must update your system to reflect the new name. There is a registry key called `Ingrian` under `\HKEY_LOCAL_MACHINE\SOFTWARE\`. If you change the name or location of the properties file, make sure that the value in the `NAE_Properties_Config` registry key reflects the path to the properties file, including the file name.

The Parameters

Once you install the SafeNet client software, you can customize it to meet the needs of your environment by modifying the properties file. The parameters listed in the file, including the delivered settings, are shown below.

```
Version=2.5  
NAE_IP=  
NAE_Port=9000  
Protocol=tcp  
  
Use_Persistent_Connections=yes  
Size_of_Connection_Pool=300  
Connection_Timeout=30000  
Connection_Idle_Timeout=600000  
Connection_Retry_Interval=600000  
Cluster_Synchronization_Delay=100  
#EdgeSecure_Name=  
  
#Cipher_Spec=HIGH:!ADH:!DH:!DSA:!EXPORT:RSA+RC4:RSA+DES:RSA+AES  
CA_File=  
Cert_File=  
Key_File=  
Passphrase=  
  
Symmetric_Key_Cache_Enabled=no  
Symmetric_Key_Cache_Expiry=43200  
  
Persistent_Cache_Enabled=no  
Persistent_Cache_Directory=  
Persistent_Cache_Expiry_Keys=43200  
Persistent_Cache_Max_Size=100  
  
Log_Level=MEDIUM  
Log_File=  
Log_Rotation=Daily  
Log_Size_Limit=100k
```

Version

The **Version** parameter indicates the version of the properties file and should not be modified.

NAE_IP

The **NAE_IP** parameter specifies the IP address of the DataSecure.

Port

The **NAE_Port** specifies the port of the DataSecure. The default port is 9000.

Important! Clients and servers must use the same port.

Protocol

The **Protocol** specifies the protocol used to communicate between the client and the DataSecure.

Possible settings:

- **tcp**
- **ssl** - The ssl option uses TLSv1. By default, TLSv1 is enabled on all DataSecures. *If you have disabled the use of TLSv1 on your servers, then you cannot establish SSL connections with between your NAE clients and servers.*

Important! Clients and servers must use the same protocol. If your DataSecures are listening for SSL requests, and your clients aren't sending SSL requests, you will run into problems.

Tip: We recommend that you gradually increase security after confirming connectivity between the client and the DataSecure. Once you have established a TCP connection between the client and server, it is safe to move on to SSL. Initially configuring a client under the most stringent security constraints can complicate troubleshooting.

Use_Persistent_Connection

The **Use_Persistent_Connections** parameter enables the persistent connections functionality.

Possible settings:

- **yes** - Enables the feature. The client establishes persistent connections with the NAE Servers. This is the default value.
- **no** - Disables the feature. A new connection is made for each connection request. The connection is closed as soon as the client receives the server response.

Size_of_Connection_Pool

The **Size_of_Connection_Pool** parameter is the total number of client-server connections that your configuration could possibly allow. (Not what actually exists at a given moment.)

Possible settings:

- **Any positive integer** - The default is 300.

Connections in the pool can be active or waiting, TCP or SSL. A connection is created as needed, and the pool scales as needed. The pool starts at size 0, and can grow to the value set here. Once the pool is full, new connection requests must wait for an existing connection to close.

Connection pooling is configured on a per-client basis. The size of the pool applies to each *client*, it is not a total value for a DataSecure or for a load balancing group. If there are multiple clients running on the same machine, separate connection pools are maintained for each client.

Connection_Timeout

The **Connection_Timeout** parameter specifies how long the client waits for the TCP connect function before timing out.

Possible settings:

- **0** - disables this setting. The client uses the operating system's connect timeout.
- **Any positive integer** - The default is 30000ms.

Setting this parameter a few hundred ms *less* than the operating system's connection timeout makes connection attempts to a downed server fail faster, and failover happens sooner. If a connection cannot be made before the timeout expires, the server is marked as down and taken out of the rotation.

Note: If your client is working with many versions of a key, do not set the **Connection_Timeout** parameter too low. Otherwise the client connection may close before the operation is complete.

Connection_Idle_Timeout

The **Connection_Idle_Timeout** parameter specifies the amount of time connections in the connection pool can remain idle before the client closes them.

Possible settings:

- **Any positive integer** - The default is 600000ms (10 min).

Important! There are two different connection timeout values: *one on the DataSecure*, and *one in the properties file*. The value of the timeout in the properties file must be less than what is set on the server. This lets the client control when idle connections are closed. Otherwise, the client can maintain a connection that is closed on the server side, which can lead to error.

Connection_Retry_Interval

The **Connection_Retry_Interval** parameter determines how long the client waits before trying to reconnect to a disabled server. If one of the DataSecures in a load balanced configuration is not reachable, the client assumes that the server is down, and then waits for the specified time period before trying to connect to it again.

Possible settings:

- **0** - This is the infinite retry interval. The disabled server will never be brought back into use.
- **Any positive integer** - The default value is 600000ms (10 minutes).

Cluster_Synchronization_Delay

The **Cluster_Synchronization_Delay** parameter specifies how long the client will wait before assuming that key changes have been synchronized throughout a cluster. After creating, cloning, importing, or modifying a key, the client will continue to use the same DataSecure appliance until the end of this delay period.

Possible settings:

- **0** - disables the function.
- **Any positive integer**. The default is 100s. You may want a higher setting for large clusters.

For example, the client sets **Cluster_Synchronization_Delay** to 100s and sends a key creation request to appliance A, which is part of a cluster. Appliance A creates the key and automatically synchronizes with rest of the cluster. The client will use only appliance A for 100 seconds - enough time for the cluster synchronization to complete. After this time period, the client will use other cluster members as before.

EdgeSecure_Name

The client uses the **EdgeSecure_Name** parameter to communicate with an EdgeSecure.

Possible settings:

- **Name of an EdgeSecure**
- **Filename** - *the first line of the file must contain the EdgeSecure name*. Using a file enables you to use the same IngrianNAE.properties file for all of your clients and still maintain unique EdgeSecure names for each.

This parameter is tier-aware to allow for failover. Three tiers are allowed: EdgeSecure_Name.1, EdgeSecure_Name.2, and EdgeSecure_Name.3. The third tier *must* be a DataSecure.

Note: This failover feature is not associated with the Multi-Tier Load Balancing feature.

Cipherspec

The **Cipher_Spec** parameter specifies which SSL/TLS protocol and encryption algorithms to use. Multiple cipher strings can be separated by colons.

For example, the value

```
HIGH : !ADH : !DH : !DSA : !EXPORT : RSA+RC4 : RSA+DES : RSA+AES
```

specifies the following:

- do NOT use anonymous Diffie-Helman (!ADH), Diffie-Helman (!DH), nor DSA (!DSA)
- use high strength ciphers, RSA+RC4, RSA+DES, and RSA+AES

Note: The default entry is commented out in the `properties` file; this is because this parameter is compiled into the client library. **You should only modify this parameter if you prefer to use some other combination of algorithms and protocols.** Modifying this parameter overrides the value in the library.

Important! If you specify some value other than the default, you must use RSA for key exchange.

CA_File

The **CA_File** parameter refers to the CA certificate that was used to sign the server certificate presented by the NAE Server to the client.

Possible settings:

- **The path and filename** - The path can be absolute or relative to your application. Don't use quotes, even if the path contains spaces.

Because all DataSecures in a clustered environment must have an identical configuration, all servers in the cluster use the same server certificate. As such, you only need to point to one CA certificate in the **CA_File** system parameter. If you do not supply the CA certificate that was used to sign the server certificate used by the DataSecures, your client applications cannot establish SSL connections with any of the servers in the cluster.

If a local CA on the DataSecure was used to sign the NAE Server certificate, you can download the certificate for the local CA, and put that certificate on the client.

Cert_File

The **Cert_File** parameter stores the path and filename of the client certificate. This is only used when your SSL configuration requires clients to provide a client certificate to authenticate to the DataSecures.

Possible settings:

- **The path and filename** - The path can be absolute or relative to your application. Don't use quotes, even if the path contains spaces. Client certificates *must* be PEM encoded.

Important! If this value is set, the certificate and private key must be present, even if the DataSecure is not configured to request a client certificate.

Key_File

The **Key_File** parameter refers to the private key associated with the client certificate specified in the **Cert_File** parameter.

Possible settings:

- **The path and filename** - The path can be absolute or relative to your application. Don't use quotes, even if the path contains spaces. The client private key must be in PEM-encoded PKCS#12 format.

Because this key is encrypted, you must use the **Passphrase** parameter so the DataSecure can decrypt it.

Important! If this value is set, the certificate and private key must be present, even if the DataSecure is not configured to request a client certificate.

Passphrase

The **Passphrase** parameter refers to the passphrase associated with the private key.

Possible settings:

- **The passphrase associated with the private key named in Key_File**

If you do NOT provide this passphrase, the client attempts to read the passphrase from standard input; this causes the application to hang.

Important! Remember that the properties file is NOT encrypted. Make sure that this file resides in a secure directory and has appropriate permissions so that it is readable only by the appropriate application or user.

Symmetric_Key_Cache_Enabled

The **Symmetric_Key_Cache_Enabled** parameter determines if the symmetric key caching feature is enabled. *Only symmetric keys can be cached.*

Possible settings:

- **no** - Key caching is disabled. Remote encryption (encryption performed on the DataSecure) is available as normal.
- **yes** - Key caching is enabled. **Protocol** must be set to ssl. (And ssl must be configured.)
- **tcp_ok** - Key caching is enabled over both tcp and ssl connections.

Symmetric_Key_Cache_Expiry

The **Symmetric_Key_Cache_Expiry** parameter determines the *minimum* amount of time that a key will remain in the client key cache.

Possible settings:

- **0** - This is the infinite timeout setting. Keys are never purged from the client cache.
- **A positive integer** - At the end of this interval, the key will be purged from the cache the next time the library is called. The default value is 43200 **seconds** (12 hours).

Persistent_Cache_Enabled

The **Persistent_Cache_Enabled** parameter determines if the persistent key caching feature is enabled.

Possible settings:

- **yes** - The feature is enabled. To enable this feature, you must also enable symmetric key caching: **Symmetric_Key_Cache_Enabled** must be set to yes or tcp_ok.
- **no** - The feature is disabled. This is the default setting.

Persistent_Cache_Directory

The **Persistent_Cache_Directory** parameter determines where ProtectApp for .NET will create the persistent cache file.

Possible settings:

- **The path to the directory that will contain the keys** - The directory must already exist. The path can be absolute or relative to your application. Don't use quotes, even if the path contains spaces.

Persistent_Cache_Expiry_Keys

The **Persistent_Cache_Expiry_Keys** parameter determines the number of seconds after which a key may be removed from the cache. To enable the persistent cache, this value must be greater than zero.

Possible settings:

- **0** - This is the infinite timeout setting. Keys are never purged from the cache.
- **Any positive integer** - At the end of this interval, the key will be purged from the cache the next time the library is called. The default value is 43200 **seconds** (12 hours).

Persistent_Cache_Max_Size

The **Persistent_Cache_Max_Size** parameter determines the maximum number of keys that can be stored in the persistent cache.

Possible settings:

- **0** - disables the feature.
- **Any positive integer** - The default value is 100 keys.

Log_Level

The **Log_Level** parameter determines the level of logging performed by the client.

Possible settings:

- **NONE** – disables client logging. We recommend that you not disable logging.
- **LOW** – only error messages are logged.
- **MEDIUM** – the client logs error messages and warnings.
- **HIGH** – the client logs error messages, warnings and informational messages. This level generates a very large number of entries and is usually reserved for debugging.

Important! The user running your client application must have permission to write to the log file, and to create new files in the directory where the log files are created.

Log_File

The **Log_File** parameter specifies a name (and possibly a path) for the log file.

Possible settings:

- **a filename** - The log will be created in the same directory as the client. The default value is Logfile.txt.
- **a path and filename** - The path can be absolute or relative to your application. Don't use quotes, even if the path contains spaces.

Use the **INGRIAN_LOGFILE_SUFFIX** environment variable to create individual log files for each client application. When the variable is set, its value is appended to the value set in the **Log_File** parameter. When a unique value is set for each client application, each client gets its own logfile.

For example, if your **Log_File** parameter is set to /foo/Logfile and your application's **INGRIAN_LOGFILE_SUFFIX** is set to app1, then that log file will be written to /foo/Logfile.app1. Then, if you set **INGRIAN_LOGFILE_SUFFIX** to app2 in a second application, that log file will appear in /foo/bar/Logfile.app2.

Similarly, the **INGRIAN_LOGFILE_PREFIX** environment variable enables you to prepend the value set in the **Log_File** parameter. To create a log file in /tmp/application1/Logfile.txt, you would set **INGRIAN_LOGFILE_PREFIX** to /tmp/application1/ and accept the **Log_File** default.

Log_Rotation

The **Log_Rotation** parameter specifies whether logs are rotated daily or once they reach a certain size.

Possible settings:

- **Daily** - Rotates logs daily. This is the default.
- **Size** - Rotates logs when they reach the size specified in **Log_Size_Limit**.

Log_Size_Limit

The **Log_Size_Limit** parameter specifies how large log files can be before they are rotated. This parameter is used only when **Log_Rotation** is set to Size.

Possible settings:

- **Any positive integer** - The default unit is bytes. You can use the suffix k (or K) for kilobytes and m (or M) for megabytes. The default value is 100k.

Reading System Properties From the Windows Registry

By default, the client reads the system properties from the properties file; however, you can configure ProtectApp for .NET to read the system properties from the system registry instead. This option is only available in Microsoft Windows environments.

When ProtectApp for .NET is launched, it searches the registry for the ConfigFilename key, which contains the location of the properties file. If the file is found, ProtectApp for .NET stores the data and uses it while the client application is running. If the properties file is not found, ProtectApp for .NET searches the registry for the individual parameters. (If no keys are found, the client application cannot run.)

For ProtectApp for .NET to read values from the registry rather than the properties file, you must rename the IngrianNAE.properties file (to hide it) and you must create the appropriate registry keys.

As part of ProtectApp for .NET, we distribute a sample registry configuration. You can use this sample, or you can manually create string values for the system values you want to set. Be aware that using the sample registry file overwrites any existing values in the registry.

Important! You do not have to create a string value for each property in the properties file; however, you must set the **Version** parameter correctly. If you are not sure what value to provide, check the properties file that shipped with ProtectApp for .NET.

Setting Properties in the Registry via the Sample Configuration

- 1 Navigate to the directory into which you installed ProtectApp for .NET. The default directory is C:\Program Files\SafeNet ProtectApp.
- 2 Open the SampleRegistryConfig.reg file in a text editor.

The sample configuration file includes only a subset of the available system properties. The following system properties are set with the values listed below:

```
"Version"="2.0"  
"NAE_IP"="192.168.200.223"  
"NAE_Port"="9000"  
"Protocol"="tcp"  
"Use_Persistent_Connections"="yes"  
"Size_of_Connection_Pool"="300"  
"Log_Level"="HIGH"
```

- 3 Modify the parameters in the file according to the needs of your deployment. Add appropriate system properties if necessary.
- 4 Save your changes and close the file.
- 5 Double-click on SampleRegistryConfig.reg to create the necessary registry keys.

Manually Setting Properties in the Registry

To set the system properties in the registry manually, create a new string value for the system properties relevant to your deployment in the following location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Ingrian\
```

For the IP address of the DataSecure, for example, you would create a new string value called NAE_IP and you would assign it a value equal to the DataSecure's IP address. Remember, if you are specifying multiple DataSecures in a load balancing group, you only need to create one registry key. Simply separate IP addresses with a colon.

Connecting to a Server

This chapter contains the following sections:

- Overview **33**
- How it Works **33**
- Related IngrianNAE.properties Parameters **34**

Overview

To connect to an individual server, you'll need to set the network configuration parameters (which control where and how messages are sent) and the connection configuration parameters (which control how long the client will wait for communication from the server).

Tip: For a quick connection test, just enter your DataSecure appliance IP in **NAE_IP** and keep the other parameters at their default settings. Run one of our sample applications (explained in Chapter 12) to confirm that the connection works. You can then change the other parameters as you configure other features.

How it Works

The following steps describe what happens when the client attempts to connect to the server for the first time.

- 1 The client creates a session. This, in turn, creates a new load balancer, which stores the **NAE_IP**, **Port**, and **Protocol** parameters and the connection timeout and interval values. If persistent connections are enabled, the load balancer also creates a connection pool.

Note: If persistent connections are not enabled, the load balancer will not create a connection pool. Connection pools enable the client to reuse existing connections if it needs them after it receives the server response. We recommend enabling persistent connections, because the performance cost of maintaining a connection pool is much less than the cost of opening a new connection for each client request.

- 2 The client requests a connection from the load balancer. Since this is the first connection request, the load balancer creates a connection. (When the client makes future requests, the load balancer will search the connection pool for existing connections before creating a new one.)
- 3 The client will wait the duration of the **Connection_Timeout** for a server response. If the server does not respond within the timeout period, or refuses the connection, the connection fails and the client ignores the server for the duration of the **Connection_Retry_Interval**. If the server responds in time, the connection is successful.
- 4 The client obtains the connection.
- 5 The client uses the connection to send a cryptographic request to the server.
- 6 The server sends the response.
- 7 The client receives the response. The load balancer keeps the connection in the pool, if persistent connections are enabled. Otherwise, it closes the connection.
- 8 The client requests another connection. If persistent connections are enabled, the load balancer searches the connection pool for an existing connection.

Related IngrianNAE.properties Parameters

The connection to an individual server uses the following parameters in the properties file:

Parameter	Description
NAE_IP	The IP address of the DataSecure device.
Port	<p>The port on which the client will communicate with the server. <i>Your client must use the same port as the server</i>, which is set on the NAE Server Settings section on the Management Console.</p> <p>Possible Settings: The server's port number. The default <i>server</i> port is 9000.</p>
Protocol	<p>The protocol used to communicate between the client and the server. <i>Your client must use the same protocol as the server</i>, which is indicated on the NAE Server Settings section on the Management Console.</p> <p>Possible Settings: tcp (default) ssl (recommended) - To configure ssl you'll need a server cert and perhaps a client certificate, depending on your configuration.</p> <p>The ssl option uses TLSv1 as the protocol here. By default, TLSv1 is enabled on all servers. If you have disabled the use of TLSv1 on your servers, then you cannot establish SSL connections between your client and server.</p>

Parameter	Description
Connection _Timeout	<p>How long the client will wait for the connection call to return a value. If a connection cannot be established before the timeout expires, then the server is marked as down and is taken out of rotation until the Connection_Retry_Interval has passed.</p> <p>Possible Settings: 30000 ms (default) - The client will wait 30 seconds.</p> <p>0 - The client will not force a timeout. The waiting period set by the client's OS still applies; the client will wait as long as the TCP stack normally waits for a connection.</p> <p>Any positive integer.</p> <p>You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used.</p> <p>If your application is time-sensitive, set this parameter a few hundred milliseconds <i>less</i> than your OS's connection timeout. This will make connections to a down server fail more quickly, in which case failover will occur sooner.</p>
Connection _Retry_Interval	<p>How long the client waits before trying to reconnect to an unavailable server.</p> <p>Possible Settings: 0 - Infinite retry interval - The client will never try to reconnect.</p> <p>Any positive integer. The default is 600000 ms (10 min).</p> <p>You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used.</p> <p>If the server or network is under a high load, a connection timeout could occur for a running sever. If your Connection_Retry_Interval is not long enough, another connection attempt will be made to the busy server - adding to its already high load.</p>

Connection Pooling

This chapter contains the following sections:

Connection Pools	36
How it Works	36
Related IngrianNAE.properties Parameters	37
Examples	38

Connection Pools

Persistent connections are connections that are used for *multiple* client requests and DataSecure responses. Once opened, persistent connections are cached in a connection pool. You set the properties that enable persistent connections, determine the maximum size of the connection pool, and control how long unused connections are maintained.

If you do not enable persistent connections, connections are closed as soon as the client receives the server response. We recommend enabling persistent connections. The performance cost of maintaining a pool is much less than the cost of opening a new connection for each request.

How it Works

The following steps describe what happens when the feature is enabled and the client attempts to connect to the server:

- 1 Your application calls for a connection to the DataSecure server.
- 2 The client checks that **Use_Persistent_Connections** is set to yes.
- 3 The client searches the connection pool for an existing connection.
- 4 The client uses an existing connection if one is available. Otherwise, it creates a new one if there is space in the pool. If there is no space the client returns an error.
- 5 The client uses the connection to communicate with the server.
- 6 When the operation is done, the connection stays in the pool and can be reused by the client. The pool closes connections that have been idle for the length of the **Connection_Idle_Timeout**.

Related IngrianNAE.properties Parameters

To use connection pools, you will have to set the following parameters in the properties file:

Parameter	Description
Use_Persistent_Connections	<p>Enables the persistent connections functionality.</p> <p>Possible Settings: yes (default) - Enables the feature</p> <p>no - Disables the feature. A new connection is made for each connection request. The connection is closed as soon as the client receives the server response.</p>
Size_of_Connection_Pool	<p>The total number of client-server connections that your configuration could possibly allow. (Not what actually exists at a given moment.) Connections in the pool can be active or waiting, tcp or ssl. A connection is created as needed and the pool scales as needed. So, your pool automatically starts at size 0, and can grow to whatever value you set here. Once the pool is full, additional connection requests must wait for an existing connection to close.</p> <p>Connection pooling is configured on a per-client basis. The size of the pool applies to <i>each</i> client, it is not the total value for an NAE Server or for a load balancing group. If there are multiple clients running on the same machine, separate connection pools are maintained for each client.</p> <p>Possible Settings: Any positive integer. The default is 300.</p> <p>Regardless of your setting, the pool will always have at least 2 connections per NAE Server.</p> <p>The larger your connection pool, the less likely your client will get a failed connection request.</p>
Connection_Idle_Timeout	<p>The time after which the client will close idle connections in the pool.</p> <p>Possible Settings: Any positive integer. The default is 600000 ms (10 min).</p> <p>Important! The DataSecure also has a connection timeout setting on the NAE Server page on the Management Console. The NAE Server value should be greater than the value set on the client. (<i>The server setting is measured in seconds.</i>) This lets the client control when idle connections are closed. Otherwise, the client can maintain a connection that is closed on the server side, which can lead to error.</p> <p>To maintain connections during load surges, use a value high enough to span the gap between peak loads.</p>

Examples

EXAMPLE 1 - THE DEFAULTS

As delivered, the IngrianNAE.properties uses the following values:

```
Use_Persistent_Connections=yes  
Size_of_Connection_Pool=300  
Connection_Idle_Timeout=600000
```

Use_Persistent_Connections: Every time the client attempts to connect to the server, it will search the connection pool for an existing connection. A new connection will be created if the connection pool is empty. An existing connection will be re-used if one is available. If the connection pool is at full capacity, the connection request will wait until an existing connection becomes available.

Size_of_Connection_Pool: No more than 300 connections can exist at one time. The size of the pool actually starts at 0 and can scale to 300 as needed. Unused connections are closed according to the **Connection_Idle_Timeout**, and the connection pool shrinks when fewer connections are needed.

Connection_Idle_Timeout: If a connection is not being used, it will be closed after 10 minutes.

Load Balancing Groups

This chapter contains the following sections:

Overview	39
How it Works	40
Related IngrianNAE.properties Parameters	41
Examples	42

Overview

A load balancing *group* is a group of DataSecure servers that the client can connect to. The *load balancer* is a client feature that determines how best to connect to the servers in the load balancing group. When *concurrent* requests are made, the load balancer determines which server to use - the goal is to distribute connections equally among the servers.

You create a load balancing group by listing multiple DataSecure IP addresses (separated by colons) in the **NAE_IP** parameter. Like this:

```
NAE_IP.1=192.168.1.10:192.168.1.11:192.168.1.12
```

The client will use the same IngrianNAE.properties file for all members of the load balancing group. If the client uses **NAE_Port** 9000, all DataSecure devices must use port 9000.

Note: We recommend that all of the devices in a load balancing group also be members of the same cluster. Clustered servers use the same port and protocol, as well as have the same keys and users. For more on clustering, see the *DataSecure Appliance User Guide*.

Important! All members of a load-balancing group must be either FIPS-compliant or non-FIPS. You cannot mix FIPS-compliant and non-FIPS servers.

How it Works

The following steps describe what happens when the client attempts to connect to the load balancing group for the first time.

- 1 The client creates a session. This in turn creates a new load balancer, which stores the **NAE_IP**, **Port**, **Protocol**, **Connection_Timeout**, and **Connection_Retry_Interval** parameters. If persistent connections are enabled, the load balancer also creates a new connection pool for each server in the load balancing group.

Note: If persistent connections are not enabled, the load balancer will not create a connection pool. Connection pools enable the client to reuse existing connections if it needs them after it receives the server response. We recommend enabling persistent connections, because the performance cost of maintaining a connection pool is much less than the cost of opening a new connection for each client request.

- 2 The client requests a connection from the load balancer. Since this is the first connection request, the load balancer chooses one of the DataSecures *at random* and creates a connection. (When the client makes future requests, the load balancer will apply the round-robin algorithm to decide which DataSecure to use.)
- 3 The client waits the duration of **Connection_Timeout** for a server response.
- 4 Server 1 does not respond within the timeout period. The client ignores server 1 for the duration of the **Connection_Retry_Interval**.
- 5 The client attempts to connect to server 2.
- 6 The client obtains the connection.
- 7 The client uses the connection to send a cryptographic request to server 2.
- 8 Server 2 sends the response.
- 9 The client receives the response. The load balancer keeps the connection in the connection pool, if persistent connections are enabled. Otherwise, the connection is closed.
- 10 The client requests another connection. The load balancer uses the round-robin algorithm to determine which DataSecure to use. If persistent connections are enabled, the load balancer searches that DataSecure's connection pool for an existing connection.

Related IngrianNAE.properties Parameters

To connect to a load balancing group, you will have to set the following parameters in the properties file:

Parameter	Description
NAE_IP.1	The NAE_IP.1 parameter holds the IP address of the DataSecure. To create a load balancing group, store <i>multiple</i> IPs here. (Separate them with a colon.) For example, NAE_IP.1=192.168.1.10:192.168.1.11:192.168.1.12
Port	The port on which the client will communicate with the DataSecure. Your client must use the same port as the DataSecure, which is set on the NAE Server Settings section on the Management Console. <i>All servers in a load balancing group must use the same port.</i> Possible Settings: The server's port number. The default server port is 9000.
Protocol	The protocol specified here is the protocol used to communicate between the client and the DataSecure. Your client must use the same protocol as the DataSecure. <i>Clients and servers must use the same protocol. All servers in a load balancing group must use the same protocol.</i> Possible Settings: tcp (default) ssl (recommended) - To configure ssl you'll need a server cert, and perhaps a client certificate, depending on your configuration.
Connection _Timeout	How long the client will wait for the TCP connection function to return a value. If a connection cannot be established before the timeout expires, then the server is marked as down and is taken out of rotation until the Connection_Retry_Interval has passed. Possible Settings: 30000 (default) - The client will wait 30 seconds. 0 - The client will not force a timeout. The waiting period set by the client's OS still applies; the client will wait as long as the TCP stack normally waits for a connection. Some value less than your OS's connection timeout. You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used. If your application is time-sensitive, set this parameter a few hundred milliseconds <i>less</i> than your OS's connection timeout parameter. This will make connections to a down server fail more quickly, in which case failover will occur sooner.

Parameter	Description
Connection_Retry_Interval	<p>How long the client waits before trying to reconnect to an unavailable server.</p> <p>Possible Settings: 0 - Infinite retry interval - The client will never try to reconnect.</p> <p>Any positive integer. The default is 600000 ms (10 min).</p> <p>You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used.</p> <p>If the server or network is under a high load, a connection timeout could occur for a running sever. If your Connection_Retry_Interval is not long enough, another connection attempt will be made to the busy server - adding to its already high load.</p>

Examples

EXAMPLE 1 - THE DEFAULTS

As delivered, the IngrianNAE.properties uses the following values:

```
Connection_Timeout=30000
Connection_Retry_Interval=600000
```

For this example, let's set **NAE_IP.1**=alpha:beta:gamma and we'll take the Connection Configuration Parameters one by one:

Connection_Timeout: The client will get an error if it takes longer than 30seconds to get a response from a server.

Connection_Retry_Interval: If the client can't reach a server before the **Connection_Timeout**, the client will take that server out of the round-robin rotation for 10 minutes (600000 milliseconds).

EXAMPLE 2 - SETTING CONNECTION_TIMEOUT

We'll still use **NAE_IP.1**=alpha:beta:gamma, but this time we'll set **Connection_Timeout**.

```
Connection_Timeout=2000
Connection_Retry_Interval=600000
```

Connection_Retry_Interval is unchanged.

Connection_Timeout: The client gets an error if it takes more than 2 seconds to get a response from a server. In the event that the server (or your network) is under a heavy load, you could get a timeout even for a *running* server. In this case, if you set **Connection_Retry_Interval** too low, you'll just end up bombarding an already overloaded server.

Multi-Tier Load Balancing

This chapter contains the following sections:

Overview	43
How it Works	44
Related IngrianNAE.properties Parameters	45
Examples	47

Overview

The multi-tier load balancing feature enables you to create *multiple levels* of load balancing groups, called tiers. When one tier is unreachable, the system fails over to the next tier. You can have a *maximum* of three tiers. You must configure the tiers in order - e.g. you can't have tier 3 without having tiers 1 and 2.

The following parameters are tier-aware, meaning that their values can vary by tier:

- **CA_File**
- **Cert_File**
- **Connection_Idle_Timeout**
- **Connection_Retry_Interval**
- **Connection_Timeout**
- **Key_File**
- **NAE_Port**
- **Passphrase**
- **Protocol**
- **Size_of_Connection_Pool**

To vary the values by tier, add the suffix *.n* to the parameter name, where *n* is the tier number. You can opt to apply one value to all tiers by omitting the *.n* suffix.

For example, to set up the port for tiers 1 and 2 and 3 you could set the following:

```
Port.1=9000  
Port.2=9000  
Port.3=7000
```

You could also do this:

```
Port=9000  
Port.3=8000
```

Because tiers 1 and 2 do not have their own settings, they would use the Port value. Tier 3 would use the Port.3 value.

You could *not* set the following:

```
Port.2=9000  
Port.3=7000
```

As there would be no setting for tier 1.

How it Works

The following steps describe what happens when the client attempts to connect to the multi-tier load balancing group for the first time.

- 1 The client creates a session. This in turn creates a new load balancer, which stores the **NAE_IP**, **Port**, **Protocol**, **Connection_Timeout**, and **Connection_Retry_Interval** parameters. If persistent connections are enabled, the load balancer also creates a new connection pool for each server in the load balancing group.
Note: If persistent connections are not enabled, the load balancer will not create a connection pool. Connection pools enable the client to reuse existing connections if it needs them after it receives the server response. We recommend enabling persistent connections, because the performance cost of maintaining a connection pool is much less than the cost of opening a new connection for each client request.
- 2 The client requests a connection from the load balancer. Since this is the first connection request, the load balancer chooses one of the DataSecures on tier 1 *at random* and creates a connection. (When the client makes future requests, the load balancer will apply the round-robin algorithm to decide which DataSecure to use.) The load balancer chooses server 1 on tier 1.
- 3 The client waits the duration of **Connection_Timeout** for server 1's response.
- 4 Server 1 does not respond within the timeout period. The client ignores server 1 for the duration of the **Connection_Retry_Interval**.

- 5 The client attempts to connect to server 2 on tier 1.
- 6 The client waits the duration of **Connection_Timeout** for server 2's response.
- 7 The client can't connect to any server on tier 1.
- 8 The client attempts to connect to a server on tier 2. The load balancer chooses one of the DataSecures on tier 2 *at random* and tries to create a connection. When the client makes future requests, the load balancer will apply the round-robin algorithm to decide which DataSecure to use. (The client will continue to use tier 2 until tier 1 is available.) The client will cycle through all of the DataSecures on all tiers.
- 9 The client obtains the connection from server 3 on tier 2.
- 10 The client uses the connection to send a cryptographic request to server 3.
- 11 Server 3 sends the response.
- 12 The client receives the response. The load balancer keeps the connection in the pool, if persistent connections are enabled. Otherwise, the connection is closed.
- 13 The client requests another connection. The load balancer uses the round-robin algorithm to determine which DataSecure to use. If persistent connections are enabled, the load balancer searches that DataSecure's connection pool for an existing connection.

Related IngrianNAE.properties Parameters

To connect to a multi-tier load balancing group, you will have to set the following parameters in the properties file:

Parameter	Description
NAE_IP.1	<p>The NAE_IP.1 parameter holds the IP address of the DataSecure. To create multiple <i>tiers</i> or servers, increment the suffix (the .1). You can create a maximum of three tiers, and each one can be a load balancing group. For example:</p> <p>NAE_IP.1=192.168.1.10:192.168.1.11:192.168.1.12</p> <p>NAE_IP.2=172.17.8.18:172.17.8.19</p> <p>NAE_IP.3=10.20.5.55:10.20.5.56</p>
Port	<p>The port on which the client will communicate with the DataSecure. Your client must use the same port as the DataSecure, which is set on the NAE Server Settings section on the Management Console. Each device in a load balancing group must use the same port, but the port <i>can</i> vary by tier. To vary ports by tier, use the .n suffix, where n is the tier number.</p> <p>Possible Settings: The server's port. The default <i>server</i> port is 9000.</p>

Parameter	Description
Protocol	<p>The protocol specified here is the protocol used to communicate between the client and the DataSecure. <i>Your client must use the same protocol as the DataSecure.</i></p> <p>Possible Settings: tcp (default) ssl (recommended) - To configure ssl you'll need a server cert, and perhaps a client certificate, depending on your configuration.</p>
Connection _Timeout	<p>How long the client will wait for the TCP connection function to return a value. If a connection cannot be established before the timeout expires, then the server is marked as down and is taken out of rotation until the Connection_Retry_Interval has passed.</p> <p>Possible Settings: 30000 (default) - The client will wait 30 seconds.</p> <p>0 - The client will not force a timeout. The waiting period set by the client's OS still applies; the client will wait as long as the TCP stack normally waits for a connection.</p> <p>Some value less than your OS's connection timeout.</p> <p>You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used.</p> <p>If your application is time-sensitive, set this parameter a few hundred milliseconds <i>less</i> than your OS's connection timeout parameter. This will make connections to a down server fail more quickly, in which case failover will occur sooner.</p>
Connection _Retry_Interval	<p>How long the client waits before trying to reconnect to an unavailable server.</p> <p>Possible Settings: 0 - Infinite retry interval - The client will never try to reconnect.</p> <p>Any positive integer. The default is 600000 ms (10 min).</p> <p>You can use abbreviations (ms, s, m, h, d) to specify the time units (milliseconds, seconds, minutes, hours, days). If you do not include an abbreviation, the default time unit (milliseconds) is used.</p> <p>If the server or network is under a high load, a connection timeout could occur for a running sever. If your Connection_Retry_Interval is not long enough, another connection attempt will be made to the busy server - adding to its already high load.</p>

Examples

EXAMPLE 1 - THE DEFAULTS

As delivered, the IngrianNAE.properties uses the following values:

```
Connection_Timeout=30000  
Connection_Retry_Interval=600000
```

For this example, let's set **NAE_IP.1**=alpha:beta:gamma, **NAE_IP.2**=psi:omega, and we'll take the Connection Configuration Parameters one by one:

Connection_Timeout: The client will get an error if it takes longer than 30seconds to get a response from a server.

Connection_Retry_Interval: If the client can't connect to a server within the **Connection_Timeout**, the client will take that server out of the round-robin rotation for 10 minutes (600000 milliseconds).

EXAMPLE 2 - SETTING CONNECTION_TIMEOUT

We'll still use **NAE_IP.1**=alpha:beta:gamma, and **NAE_IP.2**=psi:omega, but this time we'll set **Connection_Timeout**.

```
Connection_Timeout=2000  
Connection_Retry_Interval=600000
```

Connection_Retry_Interval is unchanged.

Connection_Timeout: The client will get an error if it takes more than 2 seconds to get a response from a server. In the event that the server (or your network) is under a heavy load, you could get a timeout even for a running server. In this case, if you set **Connection_Retry_Interval** too low, you'll just end up bombarding an already overloaded server.

EXAMPLE 3 - AUTOMATIC FAILOVER

You can use the following settings to ensure a speedy failover from one tier to another:

```
Connection_Timeout=600  
Connection_Retry_Interval=600
```

Here's what happens:

Connection_Timeout: The client will get an error if it can't get a connection within .01 minutes.

Connection_Retry_Interval: The client will ignore an unavailable server for .01 minutes.

This configuration is useful when testing your Multi-Tier Load Balancing setup.

Setting up SSL

This chapter provides an overview of our SSL and SSL with Client Certificate Authentication features, and provides a walkthrough of both configuration procedures.

This chapter contains the following sections:

SSL Overview	48
SSL Configuration Procedures	49
SSL Walkthrough for SafeNet Clients	52
SSL with Client Certificate Authentication Overview	56
SSL with Client Certificate Authentication Procedures	57
SSL with Client Certificate Authentication Walkthrough for DataSecure Clients	60

SSL Overview

Standard SSL communication requires a certificate that identifies the server. This certificate is signed by a certificate authority (CA) known to both the server and the client. During the SSL handshake, the server certificate is passed to the client. The client uses a copy of the CA certificate to validate the server certificate, thus authenticating the server.

While the CA can be a third-party CA or your corporate CA, you will most likely use a local CA on the DataSecure appliance. If you are not using a local CA, consult your CA documentation for instructions on signing requests and exporting certificates.

Tip: SafeNet, Inc. recommends that you increase security *only after* confirming network connectivity. You should establish a TCP connection before enabling SSL. Otherwise, an unrelated network connection mistake could interfere with your SSL setup and complicate the troubleshooting process.

To use an SSL connection when communicating with the DataSecure appliance, you must configure both the server and the client.

To configure the server, you must:

- Create a server certificate. (If you're using a cluster, each member must have its own, unique certificate.)

This may involve the following steps:

- Creating a Local CA.
 - Creating a Server Certificate Request on the Management Console.
 - Signing a Server Certificate Request with a Local CA.
- Update the NAE Server settings on the Management Console (Device, NAE Server, NAE Server).

You'll need to check **Use SSL** and select your server certificate in the **Server Certificate** field.

To configure the client, you must:

- Place a copy of the CA certificate on your client.

This may involve the following step:

- Downloading the Local CA Certificate.
- Update `IngrianNAE.properties` file as follows:
 - `Protocol=ssl`
 - `CA_File=<location and name of the CA certificate file>`

SSL Configuration Procedures

This section describes the procedures you will follow when configuring SSL. It explains the following processes:

- [Creating a Local CA](#)
- [Creating a Server Certificate Request on the Management Console](#)
- [Signing a Server Certificate Request with a Local CA](#)
- [Importing a Server Certificate to the DataSecure Appliance](#)
- [Downloading the Local CA Certificate](#)

Creating a Local CA

To create a local CA:

- 1 Log on to the Management Console as an administrator with Certificate Authorities access control.
- 2 Navigate to the Create Local Certificate Authority section on the Certificate and CA Configuration page (Security, Certificates & CAs, Local CAs).
- 3 Modify the fields as needed.
- 4 Select either *Self-signed Root CA* or *Intermediate CA Request* as the **Certificate Authority Type**.
- 5 Click **Create**.

Note: Only a local CA can sign certificate requests on the DataSecure appliance. If you are using a CA that does not reside on the DataSecure appliance you cannot use the Management Console to sign certificate requests.

Creating a Server Certificate Request on the Management Console

To create a server certificate request on the Management Console:

- 1 Log on to the Management Console as an administrator with Certificates access control.
- 2 Navigate to the Create Certificate Request section of the Certificate Configuration page (Security, Certificates & CAs, Certificates) and modify the fields as needed.
- 3 Click **Create Certificate Request**. This creates the certificate request and places it in the Certificate List section of the Certificate and CA Configuration page. The new entry shows that the **Certificate Purpose** is *Certificate Request* and that the **Certificate Status** is *Request Pending*.

Signing a Server Certificate Request with a Local CA

To sign a server certificate request with a local CA:

- 1 Log in to the Management Console as an administrator with Certificates and Certificate Authorities access controls.
- 2 Navigate to the Certificate List section on the Certificate and CA Configuration page (Security, Certificates & CAs, Certificates).

- 3 Select the certificate request and click **Properties**.
- 4 Copy the text of the certificate request. The copied text must include the header (-----BEGIN CERTIFICATE REQUEST-----) and footer (-----END CERTIFICATE REQUEST-----).
- 5 Navigate to the Local Certificate Authority List (Security, Certificates & CAs, Local CAs). Select the local CA and click **Sign Request** to access the Sign Certificate Request section.
- 6 Modify the fields as shown:
 - **Sign with Certificate Authority** - Select the CA that signs the request.
 - **Certificate Purpose** - Select *Server*.
 - **Certificate Duration (days)** - Enter the life span of the certificate.
 - **Certificate Request** - Paste all text from the request, including the header and footer.
- 7 Click **Sign Request**. This will take you to the CA Certificate Information section.
- 8 Copy the actual certificate. The copied text must include the header (-----BEGIN CERTIFICATE-----) and footer (-----END CERTIFICATE-----).
- 9 Navigate back to the Certificate List section (Security, Certificates & CAs, Certificates). Select your certificate request and click **Properties**.
- 10 Click **Install Certificate**.
- 11 Paste the actual certificate in the Certificate Response text box. Click **Save**. The Management Console returns you to the Certificate List section. The section will now show that the **Certificate Purpose** is *Server* and that the **Certificate Status** is *Active*.

The certificate can now be used as the server certificate for the NAE Server.

Importing a Server Certificate to the DataSecure Appliance

As an alternative to the certificate creation procedure outlined above, you can import a certificate to the DataSecure appliance.

To import a certificate to the DataSecure appliance:

- 1 Log in to the Management Console as an administrator with Certificates access control.
- 2 Navigate to the Import Certificate section of the Certificate and CA Configuration page (Security, Certificates & CAs, Certificates).
- 3 Select the method used to import the certificate file.
- 4 Enter the name of the file and the private key password.
- 5 Click the **Import Certificate** button.

The certificate can now be used as the server certificate for the NAE Server.

Downloading the Local CA Certificate

To download a local CA certificate from the DataSecure appliance:

- 1 Log in to the Management Console as an administrator with Certificate Authorities access control.
- 2 Navigate to the Local Certificate Authority List section of the Certificates and CA Configuration page (Security, Certificates & CAs, Local CAs).
- 3 Select the Local CA and click the **Download** button to download the file to your client. You should place the CA certificate in a secure location and modify access appropriately.

Note: Use the `CA_File` parameter in the `IngrinNAE.properties` file to indicate the name and location of the CA certificate.

SSL Walkthrough for SafeNet Clients

This walkthrough assumes the following:

- You have read the SSL overview section.
- You have configured a TCP connection between your client and the DataSecure appliance.

There are a few different ways that you could configure SSL. For example, you can use a CA that does not reside on the DataSecure appliance or you can create a new one. This walkthrough makes such decisions for you. By following these instructions, you will:

- Create a Local CA.
- Create a Certificate Request.
- Create a Server Certificate by signing the Certificate Request with the Local CA.
- Download the Local CA to the client.

Once you have completed and understood this walkthrough, you might decide to alter some of the steps to better fit your organization's policies.

To configure SSL:

- 1 Log in to the Management Console as an administrator with Certificates, Certificate Authorities, and NAE Server access controls.
- 2 Navigate to the Create Local Certificate Authority section (Security, Certificates & CAs, Local CAs). Enter the values shown below to create a new local CA. Click **Create**.

Create Local Certificate Authority		Help ?
Certificate Authority Name:	NewLocalCA	
Common Name:	NewLocalCA	
Organization Name:	Your Organization	
Organizational Unit Name:	Your Organizational Unit	
Locality Name:	Redwood City	
State or Province Name:	CA	
Country Name:	US	
Email Address:	address@email.com	
Key Size:	2048	
Certificate Authority Type:	<input checked="" type="radio"/> Self-signed Root CA CA Certificate Duration (days): 3650 Maximum User Certificate Duration (days): 3650 <input type="radio"/> Intermediate CA Request	
<div>Create</div>		

- 3 Navigate to the Create Certificate Request section (Security, Certificates & CAs, Certificates). Enter the values shown below to create a request. Click **Create Certificate Request**.

Create Certificate Request		Help ?
Certificate Name:	NewServerCert	
Common Name:	NewServerCert	
Organization Name:	Your Organization	
Organizational Unit Name:	Your Organizational Unit	
Locality Name:	Redwood City	
State or Province Name:	CA	
Country Name:	US	
Email Address:	address@email.com	
Key Size:	1024	
<div>Create Certificate Request</div>		

- 4 Select your new certificate request from the Certificate List section (right above the Create Certificate Request section). Click **Properties**. Copy the actual request (highlighted below). Include the header and footer.

Certificate Request Information
Help

Certificate Name:	NewServerCert
Key Size:	1024
Subject:	CN: NewServerCert O: Your Organization OU: Your Organizational Unit L: Redwood City ST: CA C: US emailAddress: address@email.com

```

-----BEGIN CERTIFICATE REQUEST-----
MIIB6zCCAQQCAQAwgaoxFjAUBgNVBAMTDU5ld1N1cnZ1ckN1cnQxGjAYBgNVBAoT
EVlvdXIgT3JnYU5pemF0aW9uMSEwHwYDVQQLEhhZb3VyIE9yZ2FuaXphdGlvbmFs
IFVuaXQxFTATBgNVBACjDFJ1ZHVyb2QgQ210eTELMAkGA1UECBMCQ0ExCzAJBgNV
ken9L/updosp/gY68Yv2Lx1ngbLyAFvze+eTNkMLX7ru9sGyuPwytpoXiOf8cdeD
ux1J/PpOtXDN8oEQ23ZpcHTHZMY49fZvScWVA75gyTdRZVzk9gTVoS4KBboz+tz
d1d4USpO5NkLv6QVQOjL
-----END CERTIFICATE REQUEST-----

```

Download
Install Certificate
Create Self Sign Certificate
Back

- Navigate back to the Local Certificate Authority List section (Security, Certificates & CAs, Local CAs). Select your new local CA and click **Sign Request**.
- Select **Certificate Purpose** *Server* and paste the certificate request into the **Certificate Request** field, as shown below.

Sign Certificate Request
Help

Sign with Certificate Authority: NewLocalCA (maximum 3649 days)

Certificate Purpose:
☒ Server
☐ Client

Certificate Duration (days): 3649

Certificate Request:

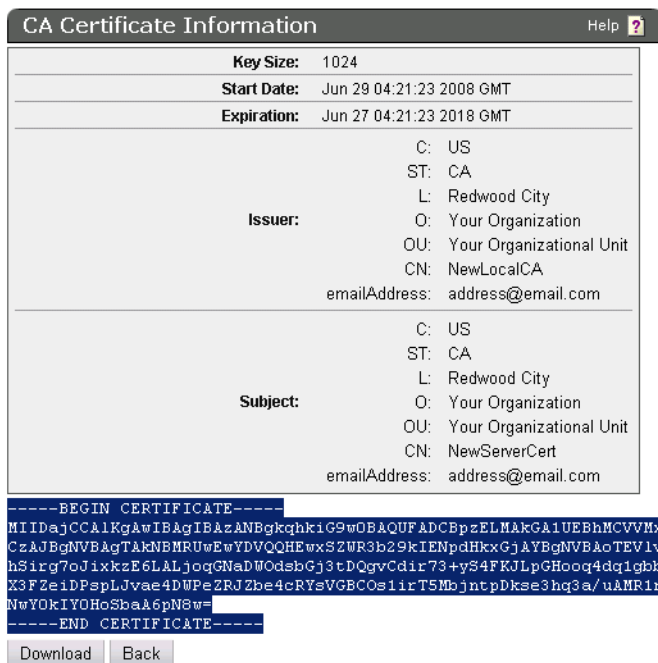
```

-----BEGIN CERTIFICATE REQUEST-----
MIIB6zCCAQQCAQAwgaoxFjAUBgNVBAMTDU5ld1N1cnZ1ckN1cnQxGjAYBgNVBAoT
EVlvdXIgT3JnYU5pemF0aW9uMSEwHwYDVQQLEhhZb3VyIE9yZ2FuaXphdGlvbmFs
IFVuaXQxFTATBgNVBACjDFJ1ZHVyb2QgQ210eTELMAkGA1UECBMCQ0ExCzAJBgNV
ken9L/updosp/gY68Yv2Lx1ngbLyAFvze+eTNkMLX7ru9sGyuPwytpoXiOf8cdeD
ux1J/PpOtXDN8oEQ23ZpcHTHZMY49fZvScWVA75gyTdRZVzk9gTVoS4KBboz+tz
d1d4USpO5NkLv6QVQOjL
-----END CERTIFICATE REQUEST-----

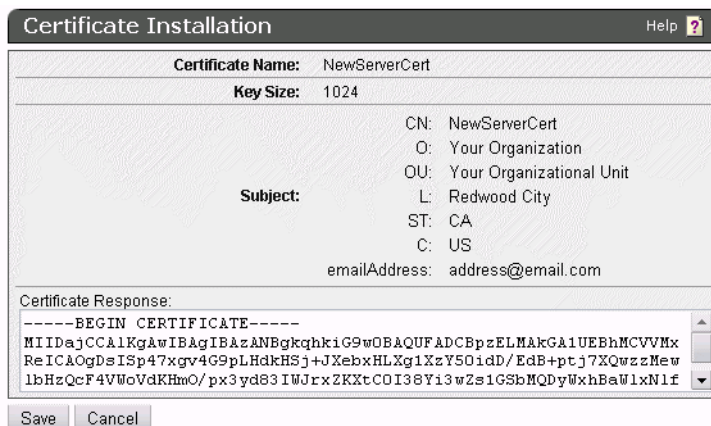
```

Sign Request
Back

- Click **Sign Request**. This will take you to the CA Certificate Information section.
- Copy the actual certificate (highlighted below). Include the header and footer.



- 9 Navigate back to the Certificate List section (Security, Certificates & CAs, Certificates). Select your certificate request and click **Properties**.
- 10 Click **Install Certificate**.
- 11 Paste the actual certificate, as shown below. Click **Save**.



The Certificate List section will now indicate that NewServerCert is an active certificate.

- 12 Navigate to the NAE Server Settings section (Device, NAE Server, NAE Server). Click **Edit**.
- 13 Check **Use SSL** and select your new server certificate in the **Server Certificate** field. Click **Save**.
- 14 Navigate back to the Local Certificate Authority List section (Security, Certificates & CAs, Local CAs). Select your new CA and click **Download**. Place the CA certificate in a secure directory on your client.
- 15 Update the following parameters in your IngrianNAE.properties file:
 - Protocol=ssl
 - CA_File=<path to CA cert>\localca.crt

You can test your configuration by running the sample.exe application.

You must create an NAE User and an encryption key on the DataSecure appliance. Then use the following command:

```
sample IngrianNAE.properties username password keyname algorithm iv  
HelloWorld
```

The DataSecure appliance will return an encrypted value for HelloWorld.

For example:

```
sample IngrianNAE.properties user1 qwerty key1 AES/CBC/PKCS5Padding  
abcdefghijklmnopqrstuvwxyz HelloWorld  
98 ed 2f 96 be 56 91 4f 20 d5 42 6f 42 e2 a6 ca
```

SSL with Client Certificate Authentication Overview

This SSL implementation requires that both the server and the client produce certificates. Each certificate is signed by a trusted CA known to both the server and the client. Most likely, you will use one CA to sign both certificates. During the SSL handshake, the certificates are exchanged. Both the client and the server use the CA certificate to validate one another's certificate, thus authenticating the other party.

► For more information about setting up SSL, see "SSL Overview" on page 48.

To enable client certificate authentication, *you must first successfully configure SSL*. Then, you must make additional configuration changes to the client and server.

Tip: We recommend that you increase security *only after* confirming network connectivity. You should establish an SSL connection before enabling Client Certificate Authentication. Otherwise, an SSL configuration mistake could interfere with your Client Certificate Authentication setup and complicate the troubleshooting process.

To configure the client, you must:

- Create a client certificate.

This may involve the following steps:

- Generating a Client Certificate Request with req.exe.
- Signing a Certificate Request and Downloading the Certificate.

You can create a certificate request using the req.exe utility or OpenSSL. You can then sign the request with the local CA on the DataSecure appliance. Once signed, the certificate request becomes a valid certificate.

If you are not using a local CA, consult your CA documentation for instructions on signing requests and exporting certificates.

- Update `IngrianNAE.properties` file as follows:
 - `Cert_File=<location and name of the client certificate>`
 - `Key_File=<location and name of the client's key file>`
 - `Passphrase=<the passphrase used to unlock the client's key file>`

To configure the server, you must:

- Place a copy of the CA certificate on your server.

This may involve the following steps:

- Installing a CA Certificate on the Server.
- Adding a CA to a Trusted CA List Profile.

- Update the NAE Server Authentication Settings section on the Management Console (Device, NAE Server, NAE Server).

You'll need to select either *Used for SSL session only* or *Used for SSL session and NAE username* in the **Client Certificate Authentication** field. The profile listed in the **Trusted CA List Profile** field must include the CA used to sign the client certificate. You can update the other fields as needed.

SSL with Client Certificate Authentication Procedures

This section describes the procedures you will follow when configuring SSL with Client Certificate Authentication. It explains the following processes:

- [Generating a Client Certificate Request with req.exe](#)
- [Signing a Certificate Request and Downloading the Certificate](#)
- [Installing a CA Certificate on the Server](#)
- [Adding a CA to a Trusted CA List Profile](#)

Generating a Client Certificate Request with req.exe

To generate a client certificate request:

- 1 Open a command prompt window and navigate to the directory where the Certificate Request Generator utility (req.exe) is installed.
- 2 Generate an RSA key and a client certificate request using the following command:

```
req -out clientreq -newkey rsa:1024 -keyout clientkey
```

where clientreq is the name of the certificate request being created, and clientkey is the name of the private key associated with the certificate request.

If you are using OpenSSL, use the following command:

```
openssl req -out clientreq -newkey rsa:1024 -keyout clientkey
```

Note: The certificate request and private key will both be created in the working directory by default. You can generate them in another directory by including a location in the request and key names. For example, to create them in the C:\certs folder, use the following command:

```
openssl req -out C:\certs\clientreq -newkey rsa:1024 -keyout  
C:\certs\clientkey
```

Note: When the FIPS mode is enabled, the private key needs to be converted into the PKCS#8 format. To convert the clientkey generated above into the PKCS#8 format, use the following command:

```
openssl.exe pkcs8 -topk8 -in clientkey -out p8clientkey -v2 des3
```

In the above command:

- **in** – Specifies the existing private key format file that is used in non-FIPS mode.
- **out** – Specifies the output PKCS#8 private key format file for use in FIPS mode.

The key generation process will then request the following data:

- A PEM passphrase to encode the private key.

The passphrase that encodes the private key is the first passphrase you provide after issuing the command above. This will be the **Passphrase** parameter in the IngrianNAE.properties file.

- The distinguished name.

The distinguished name is a series of fields whose values are incorporated into the certificate request. These fields include country name, state or province name, locality name, organization name, organizational unit name, common name, email address, surname, user ID, and IP address.

- For more information about deriving NAE usernames and authenticating client IP addresses, see “Authentication Overview” in the DataSecure Appliance User Guide.

If you will derive the NAE username from the client certificate, be sure to enter a value in the appropriate field when prompted.

If you will require client certificates to contain a source IP address, be sure to enter the IP address when prompted.

- A challenge password.
This challenge password is NOT used in the DataSecure environment.
- An optional company name.

Signing a Certificate Request and Downloading the Certificate

This section describes how to sign a certificate request with a local CA and then download the certificate. You must download the certificate *immediately* after it is signed by the CA.

To sign a certificate request with a local CA:

- 1 Open the certificate request in a text editor.
- 2 Copy the text of the certificate request. The copied text must include the header (-----BEGIN CERTIFICATE REQUEST-----) and the footer (-----END CERTIFICATE REQUEST-----).
- 3 Log in to the DataSecure appliance as an administrator with Certificate Authorities access control.
- 4 Navigate to the Local Certificate Authority List (Security, Certificates & CAs, Local CAs). Select the local CA and click **Sign Request** to access the Sign Certificate Request section.
- 5 Modify the fields as shown:
 - **Sign with Certificate Authority** - Select the CA that signs the request.
 - **Certificate Purpose** - Select *Client*.
 - **Certificate Duration (days)** - Enter the life span of the certificate.
 - **Certificate Request** - Paste all text from the request, including the header and footer.
- 6 Click **Sign Request**. This will take you to the CA Certificate Information section.
- 7 Click the **Download** button to save the certificate on your local machine. You should place the certificate in a secure location and modify access appropriately.

Note: Use the **Cert_File** parameter in the IngrianNAE.properties file to indicate the name and location of the client certificate.

Installing a CA Certificate on the Server

If the client certificate was signed by a non-local CA, you must install the CA certificate on the DataSecure appliance. To install a CA Certificate:

- 1 Log in to the DataSecure appliance as an administrator with Certificate Authorities access control.
- 2 Navigate to the Install CA Certificate section on the Certificate Authority Configuration page (Security, Certificates & CAs, Known CAs).
- 3 Enter the Certificate Name.
- 4 Paste all text from the certificate in the Certificate field, including the header and footer.
- 5 Click the **Install** button.

Adding a CA to a Trusted CA List Profile

To add the CA that signed the client certificate to the Trusted CA List Profile:

- 1 Log in to the DataSecure appliance as an administrator with Certificate Authorities access control.
- 2 Navigate to the Trusted Certificate Authority List Profiles section on the Certificate and CA Configuration page (Security, Certificates & CAs, Trusted CA Lists).
- 3 Select the profile to which you want to add the CA.
- 4 Click the **Properties** button.
- 5 Click the **Edit** button in the Trusted Certificate Authority List section.
- 6 Select the CA in the Available CAs field and click the **Add** button. This moves your CA from the Available CAs field to the Trusted CAs field.
- 7 Click the **Save** button.

Note: To enable SSL with Client Certificate Authority, the Profile containing the CA that signed the client certificate must be selected as the **Trusted CA List Profile** on the NAE Server Authentication Settings section.

SSL with Client Certificate Authentication Walkthrough for DataSecure Clients

This walkthrough assumes the following:

- You have read the SSL with Client Certificate Authentication overview section.
- You have successfully completed the SSL Walkthrough for DataSecure Clients. *You must use the Local CA created in that walkthrough.* The instructions below assume that the client and server certificates were signed by the same local CA.

There are a few different ways that you could configure SSL with Client Certificate Authentication. For example, you can use a CA that does not reside on the DataSecure appliance or you can create a new one. This walkthrough makes such decisions for you. By following these instructions, you will:

- Create a Client Certificate Request using req.exe.
- Create a Client Certificate by signing the Client Certificate Request with the Local CA on the DataSecure appliance.
- Add the Local CA to the Trusted CA List.

Once you have completed and understood this walkthrough, you might decide to alter some of the steps to better fit your organization's policies.

To configure client certificate authentication:

- 1 Open a command prompt window on the client and navigate to the directory that contains req.exe.
- 2 Generate an RSA key and a client certificate request using the following command:

```
req -out ssl\clientreq -newkey rsa:1024 -keyout ssl\clientkey
```

where the certificate request, clientreq, and the private key, clientkey, are created in the ssl directory.

If you are using OpenSSL, use the following command:

```
openssl req -out ssl\clientreq -newkey rsa:1024 -keyout  
ssl\clientkey
```

The key generation process will then request the following data:

- A PEM passphrase to encode the private key.
The passphrase that encodes the private key is the first passphrase you provide after issuing the command above. This will be the Passphrase parameter in the IngrianNAE.properties file.
- The distinguished name.
The distinguished name is a series of fields whose values are incorporated into the certificate request. These fields include country name, state or province name, locality name, organization name, organizational unit name, common name, email address, surname, user ID, and IP address.

- For more information about deriving NAE usernames and authenticating client IP addresses, see “Authentication Overview” in the DataSecure Appliance User Guide.

If you will derive the NAE username from the client certificate, be sure to enter a value in the appropriate field when prompted.

If you will require client certificates to contain a source IP address, be sure to enter the IP address when prompted.

- A challenge password.

This challenge password is NOT used in the DataSecure environment.

- An optional company name.

- 3 Open the client certificate request file and copy the actual request (highlighted below). Include the header and footer.

```
-----BEGIN CERTIFICATE REQUEST-----
MIIB6jCCAVMCAQAwZAxCAJBGNVBAYTA1VTMRMwEQYDVQQIEwppdywzZm9ybmlh
MRIwEAYDVQQHEW1QYXVIEFsdG8xEDA0BgNVBAoTB0NvbXBhbnkxFTATBgNVBAST
DENvbnBhbnkgvW5pdDENMASGA1UEAxMEdXN1c1EgMB4GC5qGSIb3DQEJARYRYWRk
Z298eG49cGp4dabtC2C2XFfmoqhG/8UEP2WxN18sQAWXCOYHFCx8yDoxq65uFcb
hPFdqyRke5Nq/XMbtAM=
-----END CERTIFICATE REQUEST-----
```

- 4 Log in to the Management Console as an administrator with Certificate Authorities access control.
- 5 Navigate to the Local Certificate Authority List section (Security, Certificates & CAs, Local CAs). Select NewLocalCA and click **Sign Request**. (NewLocalCA is the CA you created in the SSL Walkthrough.)
- 6 Select **Certificate Purpose** *Client* and paste the certificate request into the **Certificate Request** field, as shown below.

Sign Certificate Request Help ?

Sign with Certificate Authority: NewLocalCA (maximum 3649 days)

Certificate Purpose: ☐ Server ☒ Client

Certificate Duration (days): 3649

Certificate Request:

```
-----BEGIN CERTIFICATE REQUEST-----
MIIB6jCCAVMCAQAwZAxCAJBGNVBAYTA1VTMRMwEQYDVQQIEwppdywzZm9ybmlh
MRIwEAYDVQQHEW1QYXVIEFsdG8xEDA0BgNVBAoTB0NvbXBhbnkxFTATBgNVBAST
DENvbnBhbnkgvW5pdDENMASGA1UEAxMEdXN1c1EgMB4GC5qGSIb3DQEJARYRYWRk
Z298eG49cGp4dabtC2C2XFfmoqhG/8UEP2WxN18sQAWXCOYHFCx8yDoxq65uFcb
hPFdqyRke5Nq/XMbtAM=
-----END CERTIFICATE REQUEST-----
```

- 7 Click **Sign Request**. This will take you to the CA Certificate Information section.
- 8 Click **Download** to download your new client certificate (signed.crt) to your client. Place the certificate in a secure directory on your client.

- 9 Navigate to the Trusted Certificate Authority List Profiles section (Security, Certificates & CAs, Trusted CA Lists). Select **Profile Name** *Default* and click **Properties**.
- 10 Click **Edit** in the Trusted Certificate Authority List section.
- 11 Select *NewLocalCA* in **Available CAs** and click **Add**. Click **Save**.
- 12 Update the following parameters in the IngrianNAE.properties file:
 - Cert_File=<path to client cert>\client.crt
 - Key_File=<path to client key>\clientkey
 - Passphrase=<the passphrase used to unlock the client's key file>
- 13 Return to the Management Console and navigate to the NAE Server Authentication Settings section (Device Management, NAE Server) and enter the following values:
 - **Client Certificate Authentication:** Used for SSL Session only
 - **Trusted CA List Profile:** Default

Important! The CA used to sign the client certificate must be a member of the **Trusted CA List Profile**.

You can test your configuration by running the sample.exe application.

You must create an NAE User and an encryption key on the DataSecure appliance. Then use the following command:

```
sample IngrianNAE.properties username password keyname algorithm iv  
HelloWorld
```

The DataSecure appliance will return an encrypted value for HelloWorld.

For example:

```
sample IngrianNAE.properties user1 qwerty key1 AES/CBC/PKCS5Padding  
abcdefghijklmnopgh HelloWorld  
98 ed 2f 96 be 56 91 4f 20 d5 42 6f 42 e2 a6 ca
```

Creating an NAESession

This chapter contains the following sections:

Creating a Global Session to a DataSecure **64**

Creating an Authenticated Session to a DataSecure **64**

Creating a Global Session to a DataSecure

You can connect to the server by creating a session object without arguments. This creates an unauthenticated (global) session, which gives the client application the ability to create and access global keys. Whether your client can use global sessions is determined by the DataSecure settings. If password authentication is required, then global sessions are effectively disallowed.

```
NAESession * session = new NAESession();
```

Creating an Authenticated Session to a DataSecure

Creating an authenticated session is similar to the example above, except this time your application must pass in a username and password to the session object. If the username and password are valid, the client application will be authenticated and will have the ability to create keys, access keys owned by the user, and access keys available to any groups to which the user belongs.

```
NAESession * session = new NAESession(userName, password);
```


Working with Keys

This chapter contains the following sections:

- Listing All Keys Available on the DataSecure **65**
- Obtaining an Instance of a Key **65**
- Obtaining an Instance of a Key - Alternate Method **66**
- Deleting a Key Using the Key Name **66**
- Creating a Key **66**
- Importing a Symmetric Key **67**
- Importing an Asymmetric Key **68**
- Setting the Key Mode and Padding **68**

Listing All Keys Available on the DataSecure

If your client application is connected to the server via a global session, then it will only be able to see global keys; if it is connected via an authenticated session, then it will see all global keys, all keys owned by the user it is logged in as, and all keys available to any groups to which the user belongs.

```
String * sKeyNamesA[] = session->GetKeyNames();
```

This call will return an array of strings containing the key names available to the user. The server does not indicate what type of keys are being returned.

Obtaining an Instance of a Key

This example demonstrates the standard way of obtaining an instance of an AES key. To obtain an AES key, your client should create a new `NAERijndaelKey` object, passing in the session object and the name of the key on the DataSecure. This will create an instance of an `NAERijndaelKey` object that can be used to encrypt and decrypt data. If the `keyName` passed in to the DataSecure

does not match an actual key on the DataSecure, or if there is a key that matches, but that key is of a different type, an exception will be thrown and the operation will fail.

```
NAERijndaelKey * key = new NAERijndaelKey(session, keyName);
```

The key object can be used as a RijndaelKey, which is the superclass of the NAERijndaelKey class. There are similar objects for obtaining DES, DESede, RSA and HMACSHA1 keys.

Obtaining an Instance of a Key - Alternate Method

This example will demonstrate an alternate method for obtaining an instance of a key. The alternate method is a bit slower than the standard method because the NAEKey object must be cast to the appropriate key type once the instance of the key is obtained.

To get an instance of a key, create an NAEKey object by calling the GetKey method of the NAESession class on the session object, passing in the name of the key you want to access as an argument.

```
NAEKey * key = session->GetKey(keyName);
```

Deleting a Key Using the Key Name

Keys can be deleted in one of two ways. One option is to pass in the key name you want to delete; the other option is to pass in a key object instead. In both cases, you will call the DeleteKey method of the NAESession class. To delete a key by passing in the key name in a string, use the call below:

```
session->DeleteKey(keyName);
```

To delete the key by passing in a key object, use this call:

```
session->DeleteKey(key);
```

Creating a Key

In this example, we'll create a 128-bit AES key and assign permission to encrypt and decrypt to Group1; in addition, we'll make the key deletable and exportable. The same logic applies for DES, DESede, RSA, and HMACSHA1 keys; however, when assigning permissions to RSA keys, you must use the NAEAsymmetricKeyPermissions class instead of NAESymmetricKeyPermissions. Likewise, when assigning permissions to HMAC keys, you must use NAEKeyedHashPermissions.

The first step in this process is to create a new key object. The key will actually be created on the DataSecure in the last step when we call GenerateKey() and specify a name for the key. Before calling GenerateKey(), though, you can specify group permissions for the key. This is done by

creating a new permissions object and specifying which attributes you want the key to have in that permissions object. The properties of the permissions object are Group, CanEncrypt, and CanDecrypt; Group is a string value, and the other two are boolean values. The KeySize attribute comes from the RijndaelKey superclass.

```
NAERijndaelKey * key = new NAERijndaelKey(session);
key->GroupPermissions = new NAEsymmetricKeyPermissions("Group1", true,
true);
key->KeySize = 128;
key->IsDeletable = true;
key->IsExportable = true;
key->GenerateKey(keyName); // provide a name for the key here.
```

Importing a Symmetric Key

In this example, we'll import an AES key. Again, the logic to import other types of keys is similar to what is shown below, with the exception that you will use a class other than NAERijndaelKey. The example below is broken up into two sections: in the first section we create an NAEsymmetricKeyPermissions object and assign permissions. In the second section, we import the key.

In this example, we will allow Group1 to encrypt and decrypt with the key, Group2 is allowed to encrypt with the key, and Group3 is allowed to decrypt with the key. To set these permissions we create an NAEsymmetricKeyPermissions object with three members, and then we assign values to each member in the array.

Making the key deletable and exportable has been demonstrated in previous examples; what is new in this code sample is the line that actually associates the NAEsymmetricKeyPermissions object with the key object. And finally, we call the ImportKey method of the NAEsession class, passing in the name of the imported key and the bytes of the key to import.

```
NAESymmetricKeyPermissions* permsA __gc[];
permsA = new NAEsymmetricKeyPermissions* __gc[3];
permsA[0] = new NAEsymmetricKeyPermissions("group1", true, true);
permsA[1] = new NAEsymmetricKeyPermissions("group2", true, false);
permsA[2] = new NAEsymmetricKeyPermissions("group3", false, true);

NAERijndaelKey * newKey = new NAERijndaelKey(session);
newKey->IsDeletable = true;
newKey->IsExportable = true;
newKey->GroupPermissions = permsA;
newKey->KeySize = oldKey->KeySize;
newKey->ImportKey(newKeyName, oldKey->Key);
```

Importing an Asymmetric Key

You might notice that the code to import an RSA key is similar to the code above. The difference is that there are more group permissions to set because RSA keys can be used to encrypt and decrypt data and to generate signatures. In addition, when calling `ImportKey`, you must pass in an `RSAParameters` object with the key bytes. In the example below, `oldKey` is an instance of the key to import, and `newKeyName` is the name the key will have on the `DataSecure`. The `true` flag passed into the `ExportParameters` method indicates that both the public and private parts of the key will be imported. This is required.

```
NAERSAKey * newKey = new NAERSAKey(session);
newKey->IsDeletable = true;
newKey->IsExportable = true;
newKey->GroupPermissions = new NAEAsymmetricKeyPermissions("Group1",
true, true, true, true);
newKey->ImportKey(newKeyName, oldKey->ExportParameters(true));
```

The five properties of the `NAEAsymmetricKeyPermissions` object are:

- **group name** (string)
- **private key** – True if group members can use the private key to decrypt.
- **public key** – True if group members can use the public key to encrypt.
- **sign** – True if members of the group can use the key to sign data.
- **verify** – True if group members can use the key to verify signatures.

Setting the Key Mode and Padding

You can set the key mode and padding for a symmetric key (AES, DES, or DESede) for an entire session. These values are stored until the program exits. Each subsequent time you access the key, the values stored in memory are used, unless you override them when you access the key.

```
key->Mode = CipherMode::CBC;
key->Padding = PaddingMode::PKCS7;
```

Using Versioned Keys

This chapter contains the following sections:

- Overview **69**
- Creating a Versioned Key **70**
- Creating a New Version **70**
- Activate, Restrict, or Retire a Version **70**
- Using a Versioned Key to Encrypt, Sign, and MAC **70**
- Using a Versioned Key to Decrypt, SignV, and MACV **71**

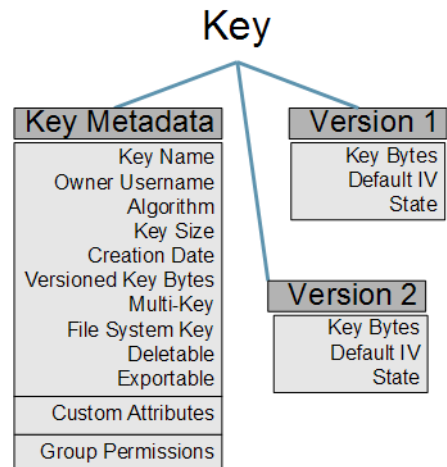
Overview

A versioned key maintains the same key metadata (key name, owner, algorithm, key size, etc), but has a unique set of bytes for each version. Thus, each version is different enough for encryption purposes, but similar enough to allow for easy management.

Each key version has its own key bytes, default IV, state, and creation date. The state determines which operations are available for a key version. Possible states are: active, restricted, and retired.

- **Active:** encryption and decryption and all key management options are allowed.
- **Restricted:** only key information operations are allowed.
- **Retired:** no operations or access to key management is allowed.

The state, combined with the key type and group permissions determine how the key version can be used. Ultimately, a key version can only be used when: the key's group permissions permit the



operation, the key version's state permits the operation, and the request comes from a member of the permitted group.

A key can have a maximum of 4000 versions.

Creating a Versioned Key

You cannot use ProtectApp for .NET to create a versioned key. You can instead create the key on the Management Console.

Creating a New Version

You cannot use ProtectApp for .NET to create a new version of a key. You can instead create the new version on the Management Console.

Activate, Restrict, or Retire a Version

You cannot use ProtectApp for .NET to alter the state of a key version. You can instead modify the key version on the Management Console.

Using a Versioned Key to Encrypt, Sign, and MAC

To encrypt, sign, and generate MACs, your code must create an instance of a key. When using a versioned key, you can create an instance of the default version, or a specific version. The code is similar for each.

You can only encrypt, sign, and generate MACs using active versions of a key.

To access the default version of a versioned key, call the same method the same way you access a non-versioned key. The DataSecure will return the latest active version.

```
// for DES key
DES* defaultKey = new NAEDESKey(session, "YourDES");

// for AES key
NAERijndaelKey * key = new NAERijndaelKey(session, "YourAES");

// for RSA key
NAERSAKey * key = new NAERSAKey(session, "YourRSA");
```

To create an instance of a specific key version, you must append a # plus the version number. These statements will return version two of the key:

```
// for DES key
DES* secVersion = new NAEDesKey(session, "YourDES#2");

// for AES key
NAERijndaelKey* secVersion = new NAERijndaelKey(session, "YourAES#2");

// for RSA key
NAERSAKey* secVersion = new NAERSAKey(session, "YourRSA#2");
```

Using a Versioned Key to Decrypt, SignV, and MACV

When data is encrypted, signed, or MACed using a versioned key, the resulting ciphertext contains information in its header indicating which version of the key was used. This header is 3 bytes long. During decryption or verification process, the DataSecure parses this information and applies the correct key version. There is no need to specify the key version.

Note: If the data requires a retired key version, you will get an exception.

Symmetric Key Caching

This chapter contains the following sections:

Overview	72
How it Works	73
Related IngrianNAE.properties Parameters	73
Logging	74

Overview

The key caching feature enables you to export symmetric keys from the DataSecure and store them on the client for a limited time in order to perform cryptographic operations locally.

Keys cached on the client are stored in process memory only, they are not stored on disk.

This feature can improve performance, specifically if network latency is high, encryption sizes are small, and local CPU cycles are available. Once keys are cached, your client's crypto operations can continue without access to the server.

Only symmetric keys (AES, DES, DESede, SEED, RC4) that have been marked *Exportable* may be cached. In addition, the user must have export privileges for the key. Thus, the user must be the key owner or the key must be global. The user automatically has full encryption and decryption privileges for all keys in the client cache; while in the cache, key permissions and authorization policies are ignored.

WARNING! Your client and its connection to the DataSecure *must* be secure. Downloading keys over this connection and storing them on your client exposes them to possible attack. When using the symmetric key caching feature, be sure that you are using a secure method of download and that your client's operating system is secure.

Supported Functions

The following functions are supported by the symmetric key cache feature (all other functions require access to the DataSecure):

- **NAESession** - Use the NAESession constructor that requires the following parameters: String* Username, String* Password, String* Passphrase.

Note: You cannot use NAESession's GetKey(String* keyName) method without access to the DataSecure. This is because the method can't determine the algorithm name without connecting to the server.

How it Works

The following steps describe what happens when the feature is enabled and the client requests a key:

- 1 The client requests a key.
- 2 The client checks if **Symmetric_Key_Cache_Enabled** is yes (or tcp_ok). If the feature is enabled, the client will search for the key in the key cache.
- 3 The client does not find the key in the cache.
- 4 The client requests the key from the server. If the user has permission and the key is exportable, the server will download the key to the client. The key is stored in the cache.
- 5 Subsequent requests for that key will utilize the key cache until the time set in **Symmetric_Key_Cache_Expiry** has passed.

Related IngrianNAE.properties Parameters

To use the symmetric key cache, you will have to set the following parameters in the properties file:

Parameter	Description
Symmetric_Key_Cache_Enabled	Enables symmetric key caching. This value must be set to yes or tcp_ok. Selecting yes enables key caching over an SSL connection, so you must also configure SSL. Selecting tcp_ok enables key caching over both tcp and ssl connections. WARNING! TCP is not a secure communication protocol.
Symmetric_Key_Cache_Expiry	The time after which a key may be removed from the symmetric key cache. The cache is only cleaned when it is used, therefore, keys may stay in the cache longer than this value. This value must be smaller than Persistent_Cache_Expiry_Keys . Otherwise, keys will be removed from the persistent key cache before they expire from the symmetric key cache.

Logging

The server will log all key downloads in the NAE log. The client will log when key caching is enabled. When **Log_Level** is set to HIGH, the client will log the following actions:

- key downloads.
- use of downloaded key.
- deletion of key from cache.
- deletion of key from cache.

Persistent Key Caching

This chapter contains the following sections:

Overview	75
Supported Functions	76
How it Works	76
Related IngrianNAE.properties Parameters	77
Logging	78
Tips	78

Overview

The persistent key cache is a secure cache on the client's disk that is used to store keys that have been downloaded from the DataSecure appliance. This cache is used when a key does not exist in the symmetric key cache and when the client can't connect to the server to access the key.

The persistent key cache can only be accessed when the correct passphrase is used. The key cache is not transferable between users, client applications, or platforms.

Note: Key caches created using v4.8.0 of a SafeNet client are not valid when using v4.8.5 of the SafeNet client. When you install v4.8.5, you must remove the old key cache and create a new one.

Unlike the symmetric key cache, which stores keys in memory, the persistent key cache is saved to disk.

Note: The key cache does not store IVs. You must store those values elsewhere.

Note: The key cache does not support versioned keys.

Note: Only symmetric keys (those based on AES, DES, DESede, SEED, and RC4 algorithms) can be cached. Keys based on RSA and HMAC-SHA1 algorithms *cannot* be cached.

Important! Downloading keys from the DataSecure appliance poses a security threat. You *must* ensure that your network is secure before using this feature. Downloading keys from the DataSecure to your client exposes them to attack.

Supported Functions

The following functions are supported by the persistent key cache feature (all other functions require access to the DataSecure):

- **NAESession** - Use the NAESession constructor that requires the following parameters: String* Username, String* Password, String* Passphrase.

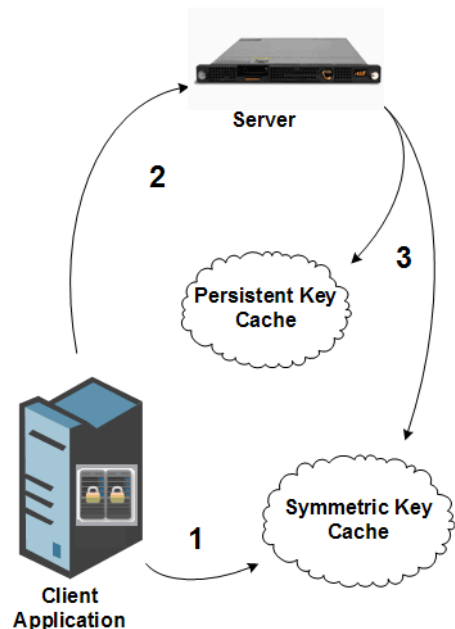
Note: You cannot use NAESession's GetKey(String* keyName) method without access to the DataSecure. This is because the method can't determine the algorithm name without connecting to the server.

How it Works

There are 2 scenarios in which the client would use the persistent key cache: when a key is downloaded to the symmetric key cache; and when the client's connection to the DataSecure is disrupted.

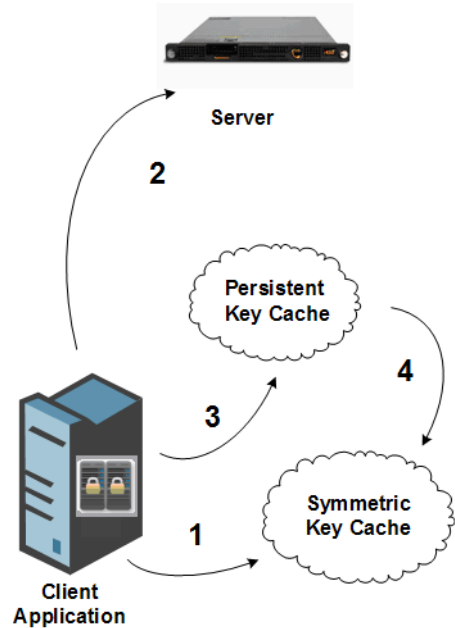
The following steps describe what happens when the feature is enabled and a key is downloaded to the symmetric key cache:

- 1 Your application requests a key. The client checks that symmetric key caching is enabled. The feature is enabled, so the client searches the symmetric key cache. The key is not found.
- 2 The client connects to the DataSecure. The server checks that the key is exportable and that the user has permission to export.
- 3 The key is exported. The client checks that the persistent key cache is enabled. The feature is enabled and the downloaded key is stored in the persistent key cache *and* the symmetric key cache.



The following steps describe what happens when the feature is enabled and the client's connection to the DataSecure is disrupted.

- 1 Your application requests a key. The client checks that symmetric key caching is enabled. The feature is enabled, so the client searches the symmetric key cache. The key is not found.
- 2 The client attempts to connect to the DataSecure. The attempt fails. The client will not attempt another connection until the **Connection_Retry_Interval** has passed. During this period, key requests that would normally be sent to the DataSecure will be sent to the persistent key cache.
- 3 The client checks that persistent key cache is enabled. The feature is enabled, so the client searches the persistent key cache for the key.
- 4 The key is found, copied to the symmetric key cache, and used. If the key was not found in the persistent key cache, the client would send an error message.



Related IngrianNAE.properties Parameters

To use the persistent key cache, you will have to set the following parameters in the properties file:

Parameter	Description
Symmetric_Key_Cache_Enabled	Enables symmetric key caching. This value must be set to yes or tcp_ok.
Symmetric_Key_Cache_Expiry	The time after which a key may be removed from the symmetric key cache. The cache is only cleaned when it is used, therefore, keys may stay in the cache longer than this value. This value must be smaller than Persistent_Cache_Expiry_Keys . Otherwise, keys will be removed from the persistent key cache before they expire from the symmetric key cache.
Persistent_Cache_Enabled	Enables the persistent key cache. This value must be set to yes.

Parameter	Description
Persistent_Cache_Directory	<p>The directory in which the persistent key cache is located. The actual cache file name uses the base name keycache, plus a suffix based on the NAE username. For example, the cache for user1 is keycache_user1. When the username contains an upper case letter, those letters are preceded by #. The cache for UserAlpha, for example, is keycache_#user#alpha. The cache for global users is keycache_no#user.</p> <p>This value must be set to an existing directory.</p>
Persistent_Cache_Expiry_Keys	<p>The time after which a key may be removed from the cache. Setting this to 0 disables the timeout; keys would not be removed from the cache. This value must be larger than Symmetric_Key_Cache_Expiry.</p>
Persistent_Cache_Max_Size	<p>Limits the number of keys in the persistent cache.</p>

Logging

The server will log all key downloads in the NAE log. The client will log the following actions:

- enabling persistent key storage.
- key downloads.
- use of downloaded key.
- deletion of key from cache.

Tips

Pre-Loading Keys

If you have advanced notice that your server will be offline, you can pre-load keys to ensure that your persistent key cache will be populated when you need it. You can use the `session.GetKey()` function to download the key to the persistent key store.

Troubleshooting

Problem	Solution
Key won't download to the cache	Check the key's exportable setting. Only exportable keys can be downloaded to the symmetric key and persistent key caches.
Keys are staying in the cache longer than the expiration maximum.	Call the library. The persistent key cache is only cleaned when it is called, therefore, keys may stay in the cache longer than the Persistent_Cache_Expiry_Keys setting.
Can't create keys when server connection is disrupted.	This works as designed. The persistent key cache is <i>not</i> used for creating keys. The client can only create keys by connecting to the DataSecure server.

Working with Certificates

This chapter contains the following sections:

Importing a Certificate	80
Exporting a Certificate	81
Exporting a CA Chain	82
Deleting a Certificate	82

Importing a Certificate

You can import a certificate to the DataSecure from your application. The DataSecure determines the certificate format from the certificate itself. When importing, set the deletable and exportable flags. If the certificate is in PKCS#12 format, include the password in the last parameter. Otherwise pass a null there.

The method call uses the following syntax:

```
NAESession.ImportCertificate (String* certificateName, bool deletable,  
bool exportable, Byte[] certificateBytes, SecureString* password);
```

Assuming the NAE session is *session*, and that *byteArray* contains the certificate, the following call imports *YourCertificate* to the DataSecure.

```
session.ImportCertificate("YourCertificate", true, true, byteArray,  
null);
```

In this example, the certificate is not in PKCS#12 format, so the password parameter contains a null.

The following call imports *YourPKCS12Certificate* to the DataSecure.

```
session.ImportCertificate("YourPKCS12Certificate", true, true, byteAr-  
ray, YourCertPassword);
```


Note: Certificates imported to the DataSecure using this method appear in the Management Console's Key and Policy Configuration page and **not** on the Certificate and CA Configuration page.

Exporting a Certificate

You have a few options when exporting a certificate from the DataSecure to your application, depending on the format of the certificate, and if you are exporting the private key. All of these options are methods of `NAESession`.

Export the Certificate Only - `ExportCertificate`

You can call `ExportCertificate()` to get the certificate in PEM format. Any private key associated with the certificate is ignored. The certificate itself must be exportable - it must have the exportable flag set to yes, otherwise you will get an error.

The method call uses the following syntax:

```
NAESession.ExportCertificate (String* certificateName);
```

Assuming the `NAESession` name is *session*, the following call puts *YourCertificate* in a byte array.

```
byteArray = session.ExportCertificate("YourCertificate");
```

Export the Certificate and the Private Key - `ExportCertificateByFormat`

Use `ExportCertificateByFormat()` to export the certificate **and** the private key from the DataSecure.

The method call uses the following syntax:

```
NAESession.ExportCertificateByFormat (String* certificateName, NAESession.NAECertificateFormat eformat, SecureString* password);
```

`NAESession.NAECertificateFormat` accepts the following values:

- `PEM_PKCS1` - requests the certificate in PEM format and the private key in PKCS#1.
- `PEM_PKCS8` - requests the certificate in PEM format and the private key in PKCS#8.
- `PKCS12` - requests both the certificate and the private key in PKCS#12 format.

Assuming the `NAESession` name is *session*, the following calls put the certificates and keys in byte arrays:

```
byteArray = session.ExportCertificateByFormat ("YourCertificate1", NAE-  
Constant.NAECertFormat.PEM_PKCS1, null);  
  
byteArray = session.ExportCertificateByFormat ("YourCertificate8", NAE-  
Constant.NAECertFormat.PEM_PKCS8, null);  
  
byteArray = session.ExportCertificateByFormat ("YourCertificate12", NAE-  
Constant.NAECertFormat.PKCS12, password);
```

Exporting a CA Chain

Exporting a CA Chain is a simple call to the ExportCaChain method.

```
byteArray = session.ExportCaChain("RootCA");
```

Deleting a Certificate

To delete a certificate from the DataSecure, call DeleteCertificate. The certificate must be flagged as deletable, which, if it wasn't done when the certificate was imported to the DataSecure, must have happened using the Management Console.

```
session.DeleteCertificate("YourCertificate");
```

Encrypting and Decrypting Data

The examples below show how to encrypt and decrypt a small piece of data. **The max data size for an individual crypto call is 168,000 bytes.**

This chapter contains the following sections:

Encrypting a String Using an AES Key	83
Decrypting a String Using an AES Key	85
Encrypting a String Using an RSA Key	86
Decrypting a String Using an RSA Key	86
Encrypting a File	87
Decrypting a File	89

Encrypting a String Using an AES Key

In this example, we encrypt a string with an AES key. The name of the key is “generic_aes_key.” Because AES is a block cipher, you must send data to the server in chunks of 16 bytes. Our string size is not a multiple of 16, so we’ll set the key to CBC mode. This will use padding to meet the 16 byte requirement. In CBC mode, the AES algorithm requires a 16 byte initialization vector (IV).

The code sample below is broken up into five sections: (1) variable declaration, (2) preparing the key object for encryption, (3) supplying text for encryption and generating an IV, (4) performing the encryption, and (5) converting the ciphertext to Base64 so that it can be printed. (The first section is self explanatory, so we won’t discuss it in detail.)

In **the second section**, we create a session object. Then get an instance of the key you want to use for the encrypt operation. In this example, the key we use is “generic_aes_key,” but you can use any AES key on your server. After obtaining an instance of the key, the key is set to CBC mode, and PKCS7 padding is specified. You can use no padding; however, this would require that the size of the data you are encrypting is some multiple of 16. Using the padding option ensures that encrypt operations will not fail because the client did not send enough data to the server.

In **the third section**, the first thing we do is create a UTF8Encoding object, which is used in the next line to convert the input string to bytes. In the third line, we create an

NAERandomNumberGenerator object, and in the fourth line, we call GetBytes on that object, passing in the NAE session object and the randomData array. Remember that the randomData array is only 16 bytes, which effectively tells the DataSecure to generate only 16 bytes of random data. These random bytes will be passed to the server during the encrypt operation as the IV. It is important that you retain these random bytes for the subsequent decrypt operation. If the IV you provide for the decrypt operation differs from what you provided for the encrypt operation, you will not be able to decrypt your ciphertext.

Note: If we were using DES or DESede instead of AES, then the IV would only be 8 bytes.

In the **fourth section**, we will perform the encrypt operation. The first step is to create a base memory stream to act as a buffer for the encrypt operation. Next, we create a CryptoStream object by calling the standard CryptoStream constructor. In this call, we will pass in the key and the IV, and set the mode to write. The third step is to do the actual encryption by calling the Write method of the CryptoStream object, passing in inputBytes for the buffer parameter, 0 as the offset, and the length of the plaintext (which is obtained by calling the Length method on inputBytes). Because we are using a small string, we only need one encrypt call.

The **fifth section** prints the ciphertext. This is optional.

```
UTF8Encoding * utf8;
CryptoStream * ostr;
unsigned char inputBytes __gc[];
MemoryStream * memstr;
Byte randomData[] = new Byte[16];

NAESession * session = new NAESession(username, password);
NAERijndaelKey * key = new NAERijndaelKey(session, "generic_aes_key");
// "generic_aes_key" must be on the server
key->Mode = CipherMode::CBC;
key->Padding = PaddingMode::PKCS7;

utf8 = new UTF8Encoding();
inputBytes = utf8->GetBytes("abcd"); // Convert string to bytes
NAERandomNumberGenerator * rng = new NAERandomNumberGenerator (session);
rng->GetBytes(session, randomData); // Get a random IV.

memstr = new MemoryStream();
ostr = new CryptoStream(memstr, key->CreateEncryptor(randomData), CryptoStreamMode::Write);
ostr->Write(inputBytes, 0, inputBytes->Length);
ostr->Close();

unsigned char tempChar __gc[] = memstr->ToArray();
String * sEncrypted = Convert::ToBase64String(tempChar);
```

The example above provides an example of one way to pass in an IV. The .NET framework provides some flexibility in this respect. If you do not want to pass in an IV, you can simply call key->CreateEncryptor(), and a random IV will be generated for you. You can then use the

line below to capture the random IV:

```
randomIV = key->IV;
```

Another way to pass in your own IV is to call:

```
key->set_IV(randomIV);
```

Decrypting a String Using an AES Key

Because the logic of the decrypt operation follows the encrypt operation so closely, we will not go into as much detail describing the code below. Instead, we will highlight a few things. Again, the code is divided into five sections. In **the first section**, we use the same variables from the encrypt operation. Remember, the randomData variable contains the IV, so do not initialize that variable.

The second section of the decrypt operation (where we obtain an instance of the key) is exactly the same as the second section of the encrypt operation.

The first two lines of **the third section** are similar between the encrypt and decrypt operations. Because the ciphertext was converted to Base64 at the end of the encrypt operation, you need to convert it to a byte array before decrypting it. We are re-using the randomData variable, which contains the IV generated before the encrypt operation. As mentioned above, if you don't pass in the exact same IV, the decrypt operation will fail.

In **the fourth section**, the lines that do the encryption and decryption are the same, except for one difference: when encrypting, you call CreateEncryptor; when decrypting, you call CreateDecryptor.

And finally, in **the fifth section**, we convert the binary data back to a string.

```
UTF8Encoding * utf8;
CryptoStream * ostr;
unsigned char inputBytes __gc[];
MemoryStream * memstr;

NAESession * session = new NAESession(username, password);
NAERijndaelKey * key = new NAERijndaelKey(session, "generic_aes_key");
key->Mode = CipherMode::CBC;
key->Padding = PaddingMode::PKCS7;

utf8 = new UTF8Encoding();
inputBytes = Convert::FromBase64String(sEncrypted);

memstr = new MemoryStream();
ostr = new CryptoStream(memstr, key->CreateDecryptor(randomData), CryptoStreamMode::Write);
ostr->Write(inputBytes, 0, inputBytes->Length);
ostr->Close();

unsigned char tempChar __gc[] = memstr->ToArray();
String *sDecrypted = new String(utf8->GetChars(tempChar));
```

Encrypting a String Using an RSA Key

This example will demonstrate how to encrypt a string with an RSA key. RSA is a stream cipher, which means that you do not provide an IV, nor do you have to send data to the DataSecure in chunks of any particular size. What you do have to take into consideration is that the RSA algorithm can only encrypt data that is the same size as the key or smaller. Because the DataSecure uses PKCS #1 padding for all RSA encrypt operations (which consumes 11 bytes of data), the maximum amount of data you can encrypt with a 512 bit (64 bytes) RSA key is 53 bytes.

The first and second sections of the code sample are not described because the concepts that underlie those sections have already been described above. **The third section** shows how to get an instance of the key that will be used to perform the encrypt operation. The key we use in this example is called "generic_rsa_key."

The fourth section does the actual encryption. The code is pretty straightforward; just call the Encrypt method of the key and pass in the data you want to encrypt. The false flag indicates that OAEP padding should not be used; this is because the DataSecure uses PKCS #1 padding instead. The encrypted bytes are converted to Base64 and stored in the variable sEncrypted, which will be used in the decrypt example later.

```
UTF8Encoding * utf8;
unsigned char inputBytes __gc[];

utf8 = new UTF8Encoding();
inputBytes = utf8->GetBytes("abcd"); // Convert input string to bytes

NAESession * session = new NAESession(username, password);
NAERSAKey * key = new NAERSAKey(session, "generic_rsa_key");
// this assumes that you will use a key called "generic_rsa_key" that is
// on the server.

String *sEncrypted = Convert::ToBase64String(key->Encrypt(inputBytes,
false));
```

Decrypting a String Using an RSA Key

The sample code for decryption with an RSA key follows the logic of the encryption quite closely. Because sEncrypted is holding a Base64 value, we must convert that back to binary bytes. Then we get an instance of the key that was used for the encryption. Like the encrypt operation, the code for decryption is quite straightforward. Just call Decrypt passing in the ciphertext and a false flag to indicate that OAEP padding should not be used.

```

UTF8Encoding * utf8;
unsigned char inputBytes __gc[];
unsigned char outputBytes __gc[];

utf8 = new UTF8Encoding();
inputBytes = Convert::FromBase64String(sEncrypted);
// Convert sEncrypted, which is the output value from encrypt.

NAESession * session = new NAESession(username, password);
NAERSAKey * key = new NAERSAKey(session, "generic_rsa_key");
// this assumes that you will use a key called "generic_rsa_key" that is
on the server.

outputBytes = key->Decrypt(inputBytes, false);
String *sDecrypted = new String(utf8->GetChars(outputBytes));

```

Encrypting a File

In previous examples, the data size was small enough that we only had to send one chunk of data to the server. In this example, though, we'll assume that the file is large enough that we need to send multiple chunks. The code sample below is broken into numbered sections. Again, the functions that have already been illustrated are not described in any detail in this example.

The list below shows how some of the more important variables are used.

- **fSource** – stream for reading plaintext data from the input file.
- **fDestination** – stream for writing ciphertext from the output file.
- **count** – used in a while loop to determine when to do the final block of encryption after the entire input file has been read.
- **inputChar** – buffer to hold plaintext data read from the input file.
- **encryptedChar** – buffer to hold the ciphertext.
- **dataSize** – the size of the data chunks sent to the DataSecure.
- **encryptedCount** – indicates how many bytes of ciphertext were produced after encrypting each chunk of data.

The first section creates a session object and obtains an instance of a key.

The second section performs a check to see if the key object is using padding. If padding is used, the ciphertext will be larger than the plaintext, which means that the size of the encrypted character buffer (dataSize) needs to increase by 16 bytes because we are using an AES key. If no padding is used, then the encrypted character buffer can be the same size as the plaintext input buffer.

The third section opens the input file and the destination file that will store the ciphertext.

The fourth section is where the actual encryption is done. Because we've structured this example so that the client will send multiple chunks of data, the code is a bit more involved than the previous examples. The first line in section 4 is where you supply a value for the IV by calling the `CreateEncryptor` method of the key object, passing in the IV as an argument. In the third line, the first chunk of data is read from the input stream and stored in the `inputChar` buffer. The number of bytes read is stored in the `count` variable. The fourth line begins a while loop with three steps: (1) encrypt the data in the input buffer, (2) write the ciphertext to the output stream, and (3) read another data chunk from the input stream.

The `TransformBlock` method performs the encryption and returns the number of bytes encrypted. The five values passed into the `TransformBlock` method are: input buffer, input offset, number of bytes to encrypt, output buffer, and output offset.

Note: The ProtectApp for .NET's buffer size limit is 150,000 bytes. To process data larger than this limit, you must make multiple calls to the `TransformBlock` method.

Next, the encrypted output buffer is written to the output stream. After the ciphertext is written out, data is once again read from the input stream. The while loop continues until all the data has been read from the input stream. Once the last block of data is read, the `TransformFinalBlock` method is called, and then the last chunk of ciphertext is written to the output stream.

The fifth section does the necessary cleanup (such as closing streams and files).

```
void encryptFile(String * inputFileNames, String * encryptedFileName, int
dataSize, Byte [] ivBytes) {
    Byte inputChar[];
    Byte encryptedChar[];
    Byte ivBytes[];
    FileStream * fSource;
    FileStream * fDestination;
    int count;
    int encryptedCount;
    ICryptoTransform * enc;

    // 1. It is assumed that the mode and padding for the key instance
    have already been set appropriately.

    NAEssion * session = new NAEssion(username, password);
    NAErijndaelKey *key = new NAErijndaelKey(session, "generic_aes_key");

    // 2. Create input & output buffers. Check if key object uses padding.
    inputChar = new Byte[dataSize];
    if (key->Padding == PaddingMode::None)
        encryptedChar = new Byte[dataSize];
    else
        encryptedChar = new Byte[dataSize+16];
```



```
// 3. Open files.
fSource = File::OpenRead(inputFileName);
fDestination = File::Create(encryptedFileName);

// 4. Do encryption.
enc = key->CreateEncryptor(ivBytes);
try {
    count = fSource->Read(inputChar, 0, dataSize);
    while (count > 0) {
        encryptedCount = enc->TransformBlock(inputChar, 0, count,
        encryptedChar, 0);
        fDestination->Write(encryptedChar, 0, encryptedCount);
        count = fSource->Read(inputChar, 0, dataSize);
    }
    encryptedChar = enc->TransformFinalBlock(inputChar, 0, count);
    // Final block
    fDestination->Write(encryptedChar, 0, encryptedChar->Length);
}
catch (NAEException * e) {
    printf("Failed to encrypt the input file.");
    exit(1);
}
// 5. Clean up
if (fSource)
    fSource->Dispose();
if (fDestination)
    fDestination->Dispose();
}
```

Decrypting a File

Because the logic of the decrypt operation follows the encrypt operation so closely, we will not go into as much detail describing each step of the code sample below. Instead, we will highlight a few things that are different. The main difference is that in the encrypt example we increased the buffer size to accommodate the larger data. When decrypting data, the plaintext will be either the same size or smaller than the ciphertext; therefore, there is no need to perform the same check that we did in the encrypt operation. Otherwise, the code sample is almost identical to the encrypt example directly above, with small exceptions, like the files you open and the arguments you pass in to the various methods. For example, instead of calling `CreateEncryptor`, you will call `CreateDecryptor`.

```

void decryptFile(String * encryptedFileName, String * outputFileName,
int dataSize, Byte [] ivBytes) {
    Byte inputChar[];
    Byte decryptedChar[];
    Byte ivBytes[];
    FileStream * fSource;
    FileStream * fDestination;
    int count;
    int decryptedCount;
    ICryptoTransform * dec;
    // 1. It is assumed that the mode and padding for the key
    // instance have already been set appropriately.
    NAESession * session = new NAESession(username, password);
    NAE RijndaelKey * newKey = new NAE RijndaelKey(session,
    "generic_aes_key");
    // 2. Create buffer. The decrypted text will be smaller than
    // the ciphertext; therefore, there's no need to perform the
    // check we did in the encrypt operation.
    inputChar = new Byte[dataSize];
    decryptedChar = new Byte[dataSize];
    // 3. Open files.
    fSource = File::OpenRead(encryptedFile);
    fDestination = File::Create(outputFileName);
    // 4. Start decryption.
    dec = key->CreateDecryptor(ivBytes);
    try {
        count = fSource->Read(inputChar, 0, dataSize);
        while ( count > 0) {
            decryptedCount = dec->TransformBlock(inputChar, 0, count,
            decryptedChar, 0);
            fDestination->Write(decryptedChar, 0, decryptedCount);
            count = fSource->Read(inputChar, 0, dataSize);
        }
        decryptedChar = dec->TransformFinalBlock(inputChar, 0, count);
        // Final block
        fDestination->Write(decryptedChar, 0, decryptedChar->Length);
    }
    catch (NAEException * e) {
        printf("Failed to decrypt the input file.");
        exit(1);
    }
    // 5. Clean up
    if (fSource)
        fSource->Dispose();
    if (fDestination)
        fDestination->Dispose();
}

```

Generating a MAC

This chapter contains the following sections:

Creating a MAC **91**

Creating a MAC

This example is similar to the other examples we've presented thus far. To use the ProtectApp for .NET to create a MAC requires that you create a session object and obtain an instance of a key. Again, we create a CryptoStream object, and we set the mode to Write. Because this stream is not being linked to another stream, we have to pass in Stream::Null. The line where the actual MAC is created is similar to previous encrypt and decrypt examples: call the Write method of the CryptoStream object and pass in the data you want to MAC. The MAC will be stored in the Hash property of the key object. In order to use the same key to create another MAC, you have to call the Initialize method of the key first.

```
NAESession * session = new NAESession(username, password);
NAEHMACSHA1 * key = new NAEHMACSHA1(session, "generic_hmac_key");
// this assumes that you will a key called "generic_hmac_key"
// that is on the server.

CryptoStream* cs = new CryptoStream(Stream::Null, key, CryptoStream-
Mode::Write);

cs->Write(data, 0, data->Length);
cs->Close();
Byte result[] = key->Hash;
key->Initialize(); // Reset.
```

Using ProtectApp for .NET API

This chapter describes the functions available in the ProtectApp for .NET, and provides deployment information you'll need to use these functions effectively.

This chapter contains the following sections:

Enabling Users to Perform Administrative Operations	92
Overview	92
Thread Safety	93
Exceptions	93
Supported Functions	93

Enabling Users to Perform Administrative Operations

Using the **Allow Key and Policy Configuration Operations** checkbox on the DataSecure page lets you limit the operations that a user can perform through the ProtectApp for .NET. This option must be *enabled* for you to create, delete, import, and export keys from ProtectApp for .NET.

Overview

The ProtectApp for .NET supports a subset of the standard .NET functions supported by Microsoft. The classes, as implemented by SafeNet, are for the most part consistent with the standard Microsoft implementation. This section provides a reference for the classes and interfaces that make up the ProtectApp for .NET. In general, you can assume that the Microsoft .NET documentation applies, unless otherwise noted. Wherever the SafeNet implementation differs from the Microsoft implementation, it is noted.

All classes in the ProtectApp for .NET are written in managed C++; you can use the same classes for VB.NET and C#, after making appropriate translations in the syntax.

All classes in ProtectApp for .NET are part of the namespace `Ingrian.Security.Cryptography`

Thread Safety

The ProtectApp for .NET complies with the Microsoft .NET specification with respect to thread safety; however you should be careful to ensure that two threads do not perform operations that modify the internal state of the same object at the same time. Take for example a scenario where two threads want to perform an encrypt operation simultaneously. As long as the two threads are not calling the same Crypto Transform at the same time, the encrypt operations will proceed normally. However, if the two threads are trying to use the same Crypto Transform simultaneously, then the two threads will not have the desired outcome.

Thread-safe:

- Thread 1: `cryptoTransform1 -> TransformBlock(...)`
- Thread 2: `cryptoTransform2 -> TransformBlock(...)`

Not Thread-safe:

- Thread 1: `cryptoTransform1 -> TransformBlock(...)`
- Thread 2: `cryptoTransform1 -> TransformBlock(...)`

Exceptions

The ProtectApp for .NET returns the standard Microsoft .NET Exceptions. If you do not understand the error returned by the application, you should check the log file. You can use the events logged by the DataSecure to help you understand the condition that caused the error.

Supported Functions

The sections below list the public functions provided by the ProtectApp for .NET. There are four types:

- Supporting Calls
- Connection Calls
- Keys-related APIs
- MAC/Hash-related APIs

Supporting Calls

This section describes the following classes:

- [NAEException](#)
- [NAERandomNumberGenerator](#)

NAEException

This is the generic exception that might be thrown by our .NET classes. You cannot create an instance of this exception, but you can catch it in a try/catch block.

This class extends `System.Security.Cryptography.CryptographicException`

NAERandomNumberGenerator

This class can be used to generate random bytes. **The largest number of random bytes you can generate is 124000.**

This class extends `System.Security.Cryptography.RandomNumberGenerator`

Constructors

Constructor	Description
<code>NAERandomNumberGenerator (NAESession* Session);</code>	Creates a new instance of the random number generator.

Methods

Method	Description
<code>static void GetBytes (NAESession* Session, unsigned char Data __gc[]);</code>	Fills the Data buffer with random bytes using the Session to make a connection to the server.
<code>static void GetNonZeroBytes (NAESession* session, unsigned char Data __gc[]);</code>	Fills the Data buffer with random non-zero bytes using the Session to make a connection to the server.
<code>virtual void GetBytes (unsigned char Data __gc[]) {GetBytes(_session, Data);}</code>	Same as superclass, but may throw an NAEException.
<code>virtual void GetNonZeroBytes (unsigned char Data __gc[]) {GetNonZeroBytes(_session, Data);}</code>	Same as superclass, but may throw an NAEException.

Connection Calls

This section describes the `NAESession` class.

NAESession

This class represents a single session to a `DataSecure`. It is used by other NAE classes to send data to and retrieve data from the server. It is also used to get keys and certificates from the `DataSecure`.

This class extends `System.Security.Cryptography.Cryptographic.IDisposable`

Constructors

Constructor	Description
<code>NAESession();</code>	Creates a new global session.
<code>NAESession(String* Username, String* Password);</code>	Creates a new session and authenticates as the username provided.
<code>NAESession(String* Username, String* Password, String* Passphrase);</code>	Creates a new session and authenticates it if possible, and sets a passphrase for the persistent store.
<code>NAESession(String* Passphrase)</code>	Creates a new and unauthenticated session and sets a passphrase for the persistent store.

Properties

Properties	Description
<code>String* Username</code>	(read) Username of the authenticated user; null if the session is global.

Methods

Method	Description
<code>String* GetKeyNames() __gc[];</code>	Retrieves a list of key names that are accessible by the authenticated user.
<code>NAEKey* GetKey(String* KeyName);</code>	Retrieves the key from the server. You must cast <code>NAEKey</code> into the appropriate key type.
<code>void DeleteKey(NAEKey* Key);</code>	Deletes the specified key from the <code>DataSecure</code> .
<code>void DeleteKey(String* KeyName);</code>	Deletes a key named <code>KeyName</code> from the server.
<code>void Dispose()</code>	Dispose of the object.
<code>void ImportCertificate (String *certName, bool deletable, bool exportable, Byte certBytes[], SecureString* password);</code>	Imports a certificate to the <code>DataSecure</code> .
<code>unsigned char ExportCertificate (String* certName) __gc[];</code>	Exports a certificate from the <code>DataSecure</code> in the default (PEM) format. This method ignores any private key associated with the certificate.
<code>unsigned char ExportCAChain (String* CaName) __gc[];</code>	Exports a CA chain from the <code>DataSecure</code> .

Method	Description
<pre> unsigned char ExportCertificateByFormat(String* certName, NAEConstant::NAECertificateFormat eformat, SecureString* password) __gc[]; void DeleteCertificate(String* certName); </pre>	<p>Exports a certificate from the DataSecure in the format specified in eformat. Available formats are:</p> <ul style="list-style-type: none"> • PEM_PKCS1 - The certificate is exported in PEM format, the private key is exported in PKCS#1 format. • PEM_PKCS8 - The certificate is exported in PEM format, the private key is exported in PKCS#8 format. • PKCS12 - The certificate and private key are exported in PKCS#12 format. <p>Deletes the certificate from the DataSecure. (This method can also be used to delete keys.)</p> <p>Note: The certificate must be flagged as deletable.</p>

Key-related APIs

This section describes the following classes:

- [NAEAsymmetricKeyPermissions](#)
- [NAEDesKey](#)
- [NAEKey](#)
- [NAEKeySpec](#)
- [NAERijndaelKey](#)
- [NAERSAKey](#)
- [NAESymmetricKeyPermissions](#)
- [NAETripleDesKey](#)

NAEAsymmetricKeyPermissions

This class represents the operations that a member of a group can perform using an RSA key.

Constructors

Constructor	Description
<pre> NAEAsymmetricKeyPermissions (String* Group, bool CanUsePrivate, bool CanUsePublic, bool CanSign, bool CanVerify) </pre>	Creates a new permission object for a group.

Properties

Properties	Description
String* Group	(read) Retrieves the group name from the permissions object.

Properties	Description
bool CanUsePrivate	(read & write) True if members of the group specified in the string immediately above can use the private key to decrypt.
bool CanUsePublic	(read & write) True if members of the group specified in the string immediately above can use the public key to encrypt.
bool CanSign	(read & write) True if members of the group specified in the string immediately above can use the key to sign data.
bool CanVerify	(read & write) True if members of the group specified in the string immediately above can use the key to verify signatures.

NAEDesKey

This class implements the DES algorithm. This class extends `System.Security.Cryptography.DES` and `System.Security.Cryptography.NAESymmetricKey`.

Constructors

Constructor	Description
<code>NAEDesKey(NAESession* Session, String* KeyName);</code>	Creates a new instance of a DES key object that accesses the key named <code>KeyName</code> on the <code>DataSecure</code> .
<code>NAEDesKey(NAESession* Session);</code>	Creates a new key object with no key bytes. This key can then be created on the <code>DataSecure</code> using <code>GenerateKey()</code> or <code>ImportKey()</code> .

Properties

Properties	Description
<code>String* KeyName</code>	(read) Key name.
<code>unsigned char Key ___gc[]</code>	(read) Retrieves the key bytes. Throws an <code>NAEException</code> if the key is not exportable. Any attempt to set this property will throw a <code>NotImplementedException</code> .
<code>int Feedback</code>	Not supported.
<code>boolean IsDeletable</code>	(read & write) True if the key can be deleted. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>boolean IsExportable</code>	(read & write) True if the key can be exported. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>NAESymmetricKeyPermissions* GroupPermissions ___gc[]</code>	(read & write) Set group permissions for the key. Throws an exception if the key already exists on the <code>DataSecure</code> .

Methods

Method	Description
<code>void GenerateKey(String* KeyName);</code>	Creates a new key on the <code>DataSecure</code> .

Method	Description
<code>void ImportKey(String* KeyName, unsigned char KeyBytes __gc[]);</code>	Imports the key to the DataSecure. KeyBytes refers to the raw key bytes of the key. If necessary, parity bits are set by this method.
<code>ICryptoTransform* CreateDecryptor();</code>	Same as superclass, but might throw an NAEException.
<code>ICryptoTransform* CreateDecryptor(unsigned char IV __gc[]);</code>	Creates a new ICryptoTransform using the current key and the specified IV. Does not modify the IV property of the instance.
<code>ICryptoTransform* CreateDecryptor(unsigned char KeyBytes __gc[], unsigned char IV __gc[]);</code>	Not supported; use CreateDecryptor() or CreateDecryptor(IV) instead.
<code>ICryptoTransform* CreateEncryptor();</code>	Same as superclass, but might throw an NAEException.
<code>ICryptoTransform* CreateEncryptor(unsigned char IV __gc[]);</code>	Creates a new ICryptoTransform using the current key and the specified IV. Does not modify the IV property of the instance.
<code>ICryptoTransform* CreateEncryptor(unsigned char KeyBytes __gc[], unsigned char IV __gc[])</code>	Not supported; use CreateEncryptor() or CreateEncryptor(IV) instead.

Important! When using DES keys with no padding, the plaintext must be greater than 0 bytes.

NAEKey

All NAE key classes implement this interface. It resides in the following namespace: System.Security.Cryptography.

Properties

Properties	Description
<code>String* KeyName</code>	(read) Retrieve the key name.
<code>String CryptoAlgName</code>	(read) Retrieves the key algorithm.
<code>bool IsExportable</code>	(read & write) True if key can be exported.
<code>bool IsDeletable</code>	(read & write) True if key can be deleted.
<code>void GenerateKey(String* KeyName)</code>	If no key named KeyName exists on the DataSecure, this method will create one.

NAEKeySpec

All NAE key classes implement this interface. It resides in the following namespace: System.Security.Cryptography.

Properties

Properties	Description
<code>int KeySize</code>	(read & write) Retrieves the key size.

Properties	Description
bool IsExportable	(read & write) True if the key can be exported.
bool IsDeletable	(read & write) True if the key can be deleted.

NAERijndaelKey

This class implements the AES algorithm. This class extends `System.Security.Cryptography.Rijndael` and `System.Security.Cryptography.NAESymmetricKey`.

Constructors

Constructor	Description
<code>NAERijndaelKey(NAESession* Session, String* KeyName);</code>	Creates a new instance of an AES key object that accesses the key <code>KeyName</code> on the <code>DataSecure</code> .
<code>NAERijndaelKey(NAESession* Session);</code>	Creates a new key object with no key bytes. This key can then be created on the <code>DataSecure</code> using <code>GenerateKey()</code> or <code>ImportKey()</code> .

Properties

Properties	Description
<code>String* KeyName</code>	(read) Key name.
<code>unsigned char Key __gc[]</code>	(read) Retrieves the key bytes. Throws an <code>NAEException</code> if the Key is not exportable. Any attempt to set this property will throw a <code>NotSupportedException</code> .
<code>int Feedback</code>	Not supported.
<code>boolean IsDeletable</code>	(read & write) True if the key can be deleted. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>boolean IsExportable</code>	(read & write) True if the key can be exported. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>NAESymmetricKeyPermissions* GroupPermissions __gc[]</code>	(read & write) Set group permissions for the key. Throws an exception if the key already exists on the <code>DataSecure</code> .

Methods

Method	Description
<code>void GenerateKey(String* KeyName);</code>	Creates a new key on the <code>DataSecure</code> .
<code>void ImportKey(String* KeyName, unsigned char KeyBytes __gc[]);</code>	Imports the key to the <code>DataSecure</code> . <code>KeyBytes</code> refers to the raw key bytes of the key. If necessary, parity bits are set by this method.
<code>ICryptoTransform* CreateDecryptor();</code>	Same as superclass, but might throw an <code>NAEException</code> .
<code>ICryptoTransform* CreateDecryptor(unsigned char IV __gc[]);</code>	Creates a new <code>ICryptoTransform</code> using the current key and the specified IV. Does not modify the IV property of the instance.

Method	Description
ICryptoTransform* CreateDecryptor (unsigned char KeyBytes __gc[], unsigned char IV __gc[])	Not supported; use CreateDecryptor() or CreateDecryptor(IV) instead.
ICryptoTransform* CreateEncryptor();	Same as superclass, but might throw an NAEException.
ICryptoTransform* CreateEncryptor (unsigned char IV __gc[]);	Creates a new ICryptoTransform using the current key and the specified IV. Does not modify the IV property of the instance.
ICryptoTransform* CreateEncryptor (unsigned char KeyBytes __gc[], unsigned char IV __gc[])	Not supported; use CreateEncryptor() or CreateEncryptor(IV) instead.

Important! When using AES keys with no padding, the plaintext must be greater than 0 bytes.

NAERSAKey

This class implements the RSA algorithm. This class extends `System.Security.Cryptography.RSA` and `System.Security.Cryptography.NAEKey`.

Constructors

Constructor	Description
NAERSAKey(NAESession* Session, String* KeyName);	Creates a new instance of an RSA key object that accesses the key named <i>KeyName</i> on the DataSecure.
NAERSAKey(NAESession* Session);	Creates a new key object with no key bytes. This key can then be created on the DataSecure using <code>GenerateKey()</code> or <code>ImportKey()</code> .

Properties

Properties	Description
String* KeyName	(read) Key name.
boolean IsDeletable	(read & write) True if the key can be deleted. Can only be set if the key does not exist on the DataSecure.
boolean IsExportable	(read & write) True if the key can be exported. Can only be set if the key does not exist on the DataSecure.
NAEAsymmetricKeyPermissions* GroupPermissions __gc[]	(read & write) Set group permissions for the key. Throws an exception if the key already exists on the DataSecure.
SignatureAlgorithm	Not supported

Methods

Method	Description
void GenerateKey(String* KeyName);	Creates a new key on the DataSecure.

Method	Description
virtual RSAParameters ExportParameters (bool includePrivateParameters);	Retrieves the key bytes. Throws an NAEException if the key is not exportable. If the boolean value includePrivateParameters is true, both the public and private parts of the key are exported. If the value is false, only the public part of the key are exported.
void ImportKey(String* KeyName, RSAParameters RSAParam);	Imports the key to the DataSecure.
void ImportKey(String* KeyName, String* xmlString);	Imports the key to the DataSecure from an XML string.
unsigned char * Decrypt(unsigned char value __gc[], bool fOAEP) __gc[]	Decrypts data. fOAEP must be false. See RSACryptoServiceProvider for more info.
unsigned char* DecryptValue(unsigned char value __gc[])	Not supported. Use Decrypt instead.
unsigned char * Encrypt(unsigned char value __gc[] bool fOAEP) __gc[]	Encrypts data. fOAEP must be false. See RSACryptoServiceProvider for more info.
unsigned char* EncryptValue(unsigned char value __gc[] bool fOAEP)	Not Supported. Use Encrypt instead.

Important! When using RSA keys, the data you encrypt must be greater than 0 bytes.

Note: The following methods from the RSACryptoServiceProvider class are not supported:

- FromXmlString
- SignData
- SignHash
- unsigned char SignData(unsigned char buffer __gc[], Object* halg) __gc[];
- bool VerifyData(unsigned char buffer __gc[], Object* halg, unsigned char signature __gc[]);
- unsigned char SignHash(unsigned char rgbHash __gc[], String* str) __gc[];
- bool VerifyHash(unsigned char rgbHash __gc[], String* str, unsigned char rgbSignature __gc[]);

Note: The following method from the RSA class is not supported:

- FromXmlString

NAESymmetricKeyPermissions

This class represents the operations that a member of a group can perform with a particular key. It resides in the following namespace: System.Security.Cryptography

Constructors

Constructor	Description
<code>NAESymmetricKeyPermissions(String* Group, bool CanEncrypt, bool CanDecrypt)</code>	Creates a new permission object for a group.

Properties

Properties	Description
<code>String* Group</code>	(read) Retrieve the group name from the permissions object.
<code>boolean CanEncrypt</code>	(read & write) True if members of the group specified in the string immediately above can encrypt using this key; false otherwise.
<code>boolean CanDecrypt</code>	(read & write) True if members of the group specified in the string immediately above can decrypt using this key; false otherwise.

NAETripleDesKey

This class implements the 3DES algorithm. This class extends `System.Security.Cryptography.TripleDES` and `System.Security.Cryptography.NAESymmetricKey`.

Constructors

Constructor	Description
<code>NAETripleDesKey(NAESession* Session, String* KeyName);</code>	Creates a new instance of a 3DES key object that accesses the key named <code>KeyName</code> on the <code>DataSecure</code> .
<code>NAETripleDesKey(NAESession* Session);</code>	Creates a new key object with no key bytes. This key can then be created on the <code>DataSecure</code> using <code>GenerateKey()</code> or <code>ImportKey()</code> .

Properties

Properties	Description
<code>String* KeyName</code>	(read) Key name.
<code>unsigned char Key __gc[]</code>	(read) Retrieves the key bytes. Throws an <code>NAEException</code> if the key is not exportable. Any attempt to set this property will throw a <code>NotImplementedException</code> .
<code>int Feedback</code>	Not supported.
<code>boolean IsDeletable</code>	(read & write) True if the key can be deleted. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>boolean IsExportable</code>	(read & write) True if the key can be exported. Can only be set if the key does not exist on the <code>DataSecure</code> .
<code>NAESymmetricKeyPermissions* GroupPermissions __gc[]</code>	(read & write) Set group permissions for the key. Throws an exception if the key already exists on the <code>DataSecure</code> .

Methods

Method	Description
<code>void GenerateKey(String* KeyName);</code>	Creates a new key named <code>KeyName</code> on the <code>DataSecure</code> .
<code>void ImportKey(String* KeyName, unsigned char KeyBytes __gc[]);</code>	Imports the key to the <code>DataSecure</code> . <code>KeyBytes</code> refers to the raw key bytes of the key. If necessary, parity bits are set by this method.
<code>ICryptoTransform* CreateDecryptor();</code>	Same as superclass, but might throw an <code>NAEException</code> .
<code>ICryptoTransform* CreateDecryptor (unsigned char IV __gc[]);</code>	Creates a new <code>ICryptoTransform</code> using the current key and the specified IV. Does not modify the IV property of the instance.
<code>ICryptoTransform* CreateDecryptor (unsigned char KeyBytes __gc[], unsigned char IV __gc[]);</code>	Not supported; use <code>CreateDecryptor()</code> or <code>CreateDecryptor(IV)</code> instead.
<code>ICryptoTransform* CreateEncryptor();</code>	Same as superclass, but might throw an <code>NAEException</code> .
<code>ICryptoTransform* CreateEncryptor (unsigned char IV __gc[]);</code>	Creates a new <code>ICryptoTransform</code> using the current key and the specified IV. Does not modify the IV property of the instance.
<code>ICryptoTransform* CreateEncryptor (unsigned char KeyBytes __gc[], unsigned char IV __gc[]);</code>	Not supported; use <code>CreateEncryptor()</code> or <code>CreateEncryptor(IV)</code> instead.

Important! When using DESede keys with no padding, the data you encrypt must be greater than 0 bytes.

MAC/Hash-related APIs

This section describes the following classes:

- [NAEKeyedHashPermissions](#)
- [NAEHMACSHA1](#)

NAEKeyedHashPermissions

This `DataSecure`-specific class represents the operations that a member of a group can perform using a particular HMAC key.

Constructors

Constructor	Description
<code>NAEKeyedHashPermissions (String* Group, bool CanMac, bool CanVerify)</code>	Creates a new permission object for a group.

Properties

Properties	Description
String* Group	(read) Retrieves the group name from the permissions object.
bool CanMac	(read & write) True if members of the group specified in the string immediately above can use the key to sign data.
bool CanVerify	(read & write) True if members of the group specified in the string immediately above can use the key to verify signatures.

NAEHMACSHA1

This class implements the HmacSHA1 algorithm. This class extends `System.Security.Cryptography.HMACSHA1` and `System.Security.Cryptography.NAEKey`.

Constructors

Constructor	Description
NAEHMACSHA1 (NAESession* Session, String* KeyName);	Creates a new instance of an HmacSHA1 key object that accesses the key names <i>KeyName</i> on the DataSecure.
NAEHMACSHA1 (NAESession* Session);	Creates a new key object with no key bytes. This key can then be created on the DataSecure using <code>GenerateKey()</code> or <code>ImportKey()</code> .

Properties

Properties	Description
String* KeyName	(read) Key name.
unsigned char Key __gc[]	(read) Retrieves the key bytes. Throws an <code>NAEException</code> if the key is not exportable. Any attempt to set this property will throw a <code>NotSupportedException</code> .
String* HashName	Not supported. Hash is fixed to DataSecure implementation.
boolean IsDeletable	(read & write) True if the key can be deleted. Can only be set if the key does not exist on the DataSecure.
boolean IsExportable	(read & write) True if the key can be exported. Can only be set if the key does not exist on the DataSecure.
NAEKeyedHashPermissions* GroupPermissions __gc[]	(read & write) Set group permissions for the key. Throws an exception if the key already exists on the DataSecure.

Methods

Method	Description
void GenerateKey (String* KeyName);	Creates a new key on the DataSecure.
void ImportKey(String* KeyName, unsigned char KeyBytes __gc[]);	Imports the key to the DataSecure. <i>KeyBytes</i> refers to the raw key bytes of the key.
void Initialize();	Overridden to throw an <code>NAEException</code> if necessary.

Notes

HmacSHA1keys can be between 128 and 256 bits. We recommend that the key size be at least 160 bits, and set the default at 160. (160 is the only size permitted when the appliance is operating in FIPS mode.)

Adhere to the following guidelines when using these properties:

- `CanReuseTransform` – should always be true.
- `CanTransformMultipleBlocks` – should always be true.
- `HashName` – not supported.
- `HashSize` – gets the value of the computed hash code in bits.
- `InputBlockSize` – should always be 1.
- `OutputBlockSize` – should always be 1.

A

authenticating clients, configuring the DataSecure 56

C

CA certificate
installing on server 60
CA certificates
clustering 26
ssl configuration 26
CanDecrypt property 102
CanEncrypt property 102
classes
NAEAsymmetricKeyPermissions 96
NAEDesKey 97
NAEException 94
NAEHMACSHA1 104
NAEKeyedHashPermissions 103
NAERandomNumberGenerator 94
NAERijndael 99
NAERijndaelKey 99
NAESession 94
NAETripleDesKey 99
client certificate authentication 26
configuration 56
clustering
CA certificates 26
connection pooling
configuration 24
content restrictions 12
CreateDecryptor method 98, 99, 103
CreateEncryptor method 98, 100, 103
cryptographic operations
asymmetric encryption 12
digital signatures 12
MAC 12
MAC Verify 12
random number generation 12
symmetric encryption 12
CryptoAlgName property 98

D

data
restrictions 12
DataSecure, authenticating clients 56
Decrypt method 101

DecryptValue method 101

E

Encrypt method 101
EncryptValue method 101
exceptions 93

G

GenerateKey method 97, 98, 99, 103
GroupPermissions property 99, 102
GroupPermissionsproperty 97

H

hardware requirements 11

I

ImportKey method 98, 99, 101, 103
IngrianNAE.properties
testing your installation 20
installation
uninstalling the ProtectApp for .NET 14
Interface
NAEKey 97
IsDeletable 97, 99, 102
IsDeletable property 98
IsExportable 97, 98, 99, 102

K

Key property 97, 99, 102
KeyName property 97, 98, 99, 102

L

logging
errors 93

M

methods
CreateDecryptor 98, 99, 103
CreateEncryptor 98, 100, 103
Decrypt 101
DecryptValue 101
Encrypt 101
EncryptValue 101
GenerateKey 97, 98, 99, 103
ImportKey 98, 99, 101, 103

N

- NAEAsymmetricKeyPermissions class 96
- NAEDesKey class 97
- NAEException class 94
- NAEHMACSHA1 class 104
- NAEKey Interface 97
- NAEKeyedHashPermissions class 103
- NAERandomNumberGenerator class 94
- NAERijndaelKey class 99
- NAESession class 94
- NAETripleDesKey class 99

P

- parameters
 - Size_of_Connection_Pool 24
- performance
 - connection timeout 24
 - timeout configuration 24
- persistent connection configuration 23
- properties
 - CanDecrypt 102
 - CanEncrypt 102
 - CryptoAlgName 98
 - GroupPermissions 97, 99, 102
 - IsDeletable 97, 98, 99, 102
 - IsExportable 97, 98, 99, 102
 - Key 97, 99, 102
 - KeyName 97, 98, 99, 102
- protocol configuration 23

R

- requirements
 - hardware and software 11

S

- Size_of_Connection_Pool parameter 24
- software requirements 11
- SSL
 - client certificate 26
 - client private key passphrase 27
 - server CA certificate 26
- System Properties
 - client certificate authentication 26
 - client private key passphrase 27
 - connection pooling 24
 - connection retry 25
 - persistent connections 23
 - protocol configuration 23
 - reading from the registry 30
 - setting 22
 - timeout configuration 24

T

- testing your installation
 - IngrianNAE.properties file 20
- Timeout Configuration 24
- trusted CA list profile 60

U

- uninstalling the ProtectApp for .NET 14

W

- Windows Registry 22
 - reading system properties from 30
 - sample configuration 31