

Nexus Personal

Technical Description

Nexus Personal: Technical Description

Publication date 2013-10-31

Copyright © 2013 Technology Nexus AB

Nexus endeavors to ensure that the information in this document is correct and fairly stated, but does not accept liability for any error or omission. The development of Nexus products and services is continuous and published information may not be up to date. It is important to check the current position with Nexus. This document is not part of a contract or license save insofar as may be expressly agreed. Nexus has been applied as a trademark of Nexus. All other trademarks are the property of their respective owners.

Table of Contents

Introduction	ix
About this Document	ix
Screendumps	ix
Product Overview	ix
Web Browser Plug-Ins	ix
Cryptographic APIs	x
Administration	x
NetDetacher	x
Product Structure	xi
Personal Process	xi
Browser Process	xi
Third-Party Applications	xii
GUI Branding	xii
Environment	xii
What is New in this Version	xii
Where to Find the Information	xii
Release.txt	xiii
Help	xiii
How to Contact Us	xiii
1. Functional Description Overview	1
Architecture	1
WebSigner	2
Signer2	3
Authentication	3
Registration Utility	3
Administration Plug-in	4
Version Plug-in	4
LogoutTokens Plug-in	4
Cryptographic APIs	4
Microsoft CSP	5
PKCS#11	5
Installation and Updating	5
Soft Token Migration	6
Integration with Standard Products	7
Integration with Internet Explorer	7
Integration with Mozilla-Based Browsers	7
Branding	7
Branding on Windows platforms	7
Branding on Mac OS X	7
Branding on Linux	7
Card Reader Support	7
2. WebSigner Plug-in	8
Introduction	8
Plug-in Activation	8
Internet Explorer	8
Mozilla-Based Browsers	9
Parameters	9
Scripting	12
Usage and GUI	13
Sample Web Pages	13
Digital Signature Format	15
3. Signer2 Plug-in	17
Introduction	17
Plug-in Activation	17
Internet Explorer	17

Mozilla-Based Browsers	17
Parameters	18
Scripting	20
Sample Web Pages	21
Internet Explorer	21
Mozilla-based browsers	21
Signer2 Signature Sample	22
Error Codes	23
4. Authentication Plug-in	25
Introduction	25
Plug-in Activation	25
Internet Explorer	25
Mozilla-Based Browsers	26
Parameters	26
Scripting	28
Sample Web Pages	29
Internet Explorer	29
Mozilla-based browsers	29
Authentication Signature Sample	30
Error codes	31
5. Registration Utility Plug-in	33
Introduction	33
Plug-in Activation	33
Internet Explorer	33
Mozilla-Based Browsers	34
Parameters	34
Scripting	35
Usage and GUI	38
Creating a Token	38
Sample Web Pages	39
Internet Explorer	39
Mozilla-based browsers	40
Error Codes	41
Format of a CMC Request and Response	42
Request	42
One Time Password	42
Response	42
6. Administration Plug-in	43
Introduction	43
Plug-in Activation	43
Internet Explorer	43
Mozilla-Based Browsers	44
Parameters	44
Scripting	45
Sample Web Pages	47
Internet Explorer	47
Mozilla-based browsers	47
Error codes	48
Configuration	49
7. LogoutTokens Plug-in	50
Introduction	50
Plug-in Activation	50
Internet Explorer	50
Mozilla-Based Browsers	51
Parameters	51
Scripting	51
Usage	51
Sample Web Pages	52

Security Issues	53
8. Version Plug-in	54
Introduction	54
Plug-in Activation	54
Internet Explorer	54
Mozilla-Based Browsers	55
Parameters	55
Scripting	55
Sample Web Pages	56
Output format	57
9. Personal PKCS#11	59
Introduction	59
Mozilla Browsers	59
PKCS #11 API	59
General Purpose	59
Slot and Token Management	59
Session Management	60
Object Management	60
Encryption and Decryption	60
Message Digesting	60
Signing and Verifying	61
Key Management	61
Random Number Generation	61
Interoperability	61
PKCS#11 Configuration	62
10. Personal CSP	63
Introduction	63
CSP Information	63
CSP Functions	63
CSP Connection	63
Key Management	63
Hashing and Digital Signatures	64
Encryption	64
Interoperability	64
Additional Comments	66
Using XEnroll with Personal CSP	66
Container Name	66
Example	67
CSP Configuration	68
11. Installation on Windows	69
Program Requirements	69
The Installation Program	69
Packaging	69
Installation Conditions	69
Installation Configuration	70
Installation Options	70
Messages from the Installation Program	70
Shortcuts	71
Installation Directory Tree	71
Web Installation Flowchart	71
Limitations	74
CSP & PKCS#11	74
Plug-ins and ActiveX Controls	75
Upgrade and Migration	75
Uninstall	75
Controlling the Behavior of Winlogon	75
Event Log and Return Codes	75
12. Installation on Macintosh	78

Introduction	78
Install	78
Uninstall	78
13. Installation on Linux	79
Introduction	79
Install	79
Uninstall	79
14. Installation on Citrix	80
Introduction	80
Install	80
15. Administration	81
Personal GUI	81
Administration GUI	81
Import and Export	82
Internal Store	82
Import Soft Tokens	82
Export Soft Tokens	83
Managing PIN Codes	83
Searching for Soft Token	84
Web Browser and Language Settings	85
Language	86
Using Personal in the Browser	86
Controlling Secure Sessions in the Browser	86
Advanced	87
Logging	87
Operating System	87
Installed Web Browsers	87
Troubleshooting Password Dialogs	87
Card Readers	87
Tray Icon	88
About	89
Help	89
Abbreviations	90
References	91
A. Eolas Patent	92
Introduction	92
Solution 1	92
Solution 2	92
B. Key Generation	93
Software Key Generation	93
C. Internal Stores	94
Background	94
File Formats	95
MAC Calculation	98
D. CSP and PKCS#11 Configuring	99
Platform dependencies	99
Implementation	99
E. PIN-Related Issues	100
PIN Caching	100
PIN Caching Mode	100
PIN Non-Caching Mode	100
Configuration Details	101
Force Login Before Sign	102

List of Figures

1. Product Structure	xi
1.1. Architecture	1
1.2. WebSigner	2
1.3. Registration	4
1.4. Authentication	5
1.5. Soft Token Migration	6
11.1. Web Installation Flowchart	72
15.1. Administration GUI	81
15.2. Import Soft Tokens	82
15.3. Export Soft Tokens	83
15.4. Managing PIN Codes	84
15.5. Searching for Soft Token	85
15.6. Web Browser and Language Settings	86
15.7. Advanced	87
15.8. Card Readers	88
15.9. About	89

List of Examples

2.1. Example of an ActiveX control activation	8
2.2. Example of a script to activate the plug-in	8
2.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag	9
2.4. Example of a script to activate the plug-in	9
2.5. Example of Direct Activation using Internet Explorer (Windows only)	13
2.6. Example of Scripting using Internet Explorer (<i>Windows only</i>)	14
2.7. Example of Direct Activation using Mozilla-based browsers	14
2.8. Example of Scripting using Mozilla-based browsers	15
3.1. Example of an ActiveX control activation	17
3.2. Example of a script to activate the plug-in	17
3.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag	18
3.4. Example of a script to activate the plug-in	18
4.1. Example of an ActiveX control activation	25
4.2. Example of a script to activate the plug-in	25
4.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag	26
4.4. Example of a script to activate the plug-in	26
5.1. Example of an ActiveX control activation	33
5.2. Example of a script to activate the plug-in	33
5.3. Example of how to activate the Mozilla-based browsers plug-in using the <OBJECT> tag	34
5.4. Example of a script to activate the plug-in	34
5.5. Example of creating a certificate request (<i>Windows only</i>)	39
5.6. Example of storing the certificate response (<i>Windows only</i>)	40
5.7. Example of creating a certificate request	40
5.8. Example of storing the certificate response	41
6.1. Example of an ActiveX control activation	43
6.2. Example of a script to activate the plug-in	43
6.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag	44
6.4. Example of a script to activate the plug-in	44
7.1. Example of an ActiveX control activation	50
7.2. Example of a script to activate the plug-in	50
7.3. Example of how to activate the Mozilla-based browsers plug-in using the <OBJECT> tag	51
7.4. Example of a script to activate the plug-in:	51
7.5. Example of how to Detect and Activate the Plug-in in Internet Explorer (<i>Windows only</i>)....	52
7.6. Example of how to Detect and Activate the Plug-in in Mozilla-based browsers	53
8.1. Example of an ActiveX control activation	54
8.2. Example of a script to activate the plug-in	54
8.3. Example of how to activate the Mozilla-based browsers plug-in using the <OBJECT> tag	55
8.4. Example of a script to activate the plug-in	55
8.5. Example of Direct Activation using Internet Explorer (<i>Windows only</i>):	56
8.6. Example of Scripting using Internet Explorer (<i>Windows only</i>)	56
8.7. Examples of Direct Activation using Mozilla-based browsers	56
8.8. Example of Scripting using Mozilla-based browsers	57
8.9. Example of a version string	58
11.1.	73
11.2.	73
11.3.	74
11.4.	74

Introduction

About this Document

There are three different platform oriented versions of Personal:

- Nexus Personal for the Windows platform
- Nexus Personal for the Mac OS X platform.
- Nexus Personal for the Linux Ubuntu platform.

As far as it has been possible, they have the same functionality. All three versions are described in this manual.

However, some functionality is only available on the Windows platform. The following functions belong to this category:

- Cryptographic Service Provider (CSP)
- ActiveX controls
- Internet Explorer
- CryptoAPI

Warning

Whenever these functions are mentioned throughout this manual, they will refer to the Windows platform only even though this may not be stated explicitly in the text. Other functions or options that only apply to one of the platforms will be highlighted with remarks like “Windows only”, “MAC OS X only” or “Linux only”.

Screendumps

Most screenshots in this manual are taken from Personal on one Windows platform. The layout may look different on other Windows platforms and on the Mac OS X and Linux platforms, but the message conveyed by the images should be clear to all users.

Product Overview

Personal is a unique software product that brings support for certificate enrollment, digital signatures and authentication to standard Internet software.

Personal includes the following set of functionality, which is described in the following section:

- “Web Browser Plug-Ins”
- “Cryptographic APIs”
- “Administration”
- “NetDetacher”

Web Browser Plug-Ins

Personal provides a set of web browsers plug-ins, which are all implemented as ActiveX controls for use with Internet Explorer and NPAPI plug-ins for Mozilla-based browsers.

- Personal WebSigner is a plug-in for digital signatures. The plug-in provides the possibility to digitally sign transactions.
- Personal Administration plug-in is used to manage tokens in Personal. It provides functions for export, import and deletion of tokens as well as administration of PINs.
- Personal Registration Utility plug-in allows a user to connect to a Certification Authority to request certificates and to store them on a software token.
- Personal Version plug-in can be used to return the installed Personal components and their versions. By using the Version plug-in, existing versions of Personal can be detected, in order to initiate a possible upgrade installation.
- Personal LogoutTokens plug-in logs out the current sessions to the tokens. Hence, it is possible to control the SSL logout process at the browser from a web server script.

Note

Plug-in is used as a common term for ActiveX controls as well as NPAPI plug-ins throughout this document.

Cryptographic APIs

- Personal PKCS#11 is an implementation of the RSA standard PKCS#11 that provides cryptographic functions and token support. Personal PKCS#11 is optimized for use with SSL in Mozilla-based browsers to access secure web sites.
- Personal Cryptographic Service Provider (CSP). The CSP is registered with the operating system and it allows use of tokens through Microsoft CryptoAPI (MSCAPI). Personal CSP is optimized for use with SSL in Internet Explorer to access secure web sites.

Administration

Personal is equipped with a graphical user interface that is used for the following administration tasks:

- Import/export of soft tokens
- Searching for disk drives with soft tokens
- Introduction
- Display and renaming of soft tokens
- PIN code management
- Web browser settings
- Diagnostics
- Language settings

NetDetacher

The purpose of NetDetacher is to control the SSL sessions and to detect and take appropriate actions when a token is removed. The token could be a smart card being removed from a card reader or a soft token residing on a memory stick. If the private key on the token has been used by a browser during an SSL session, that browser should be terminated.

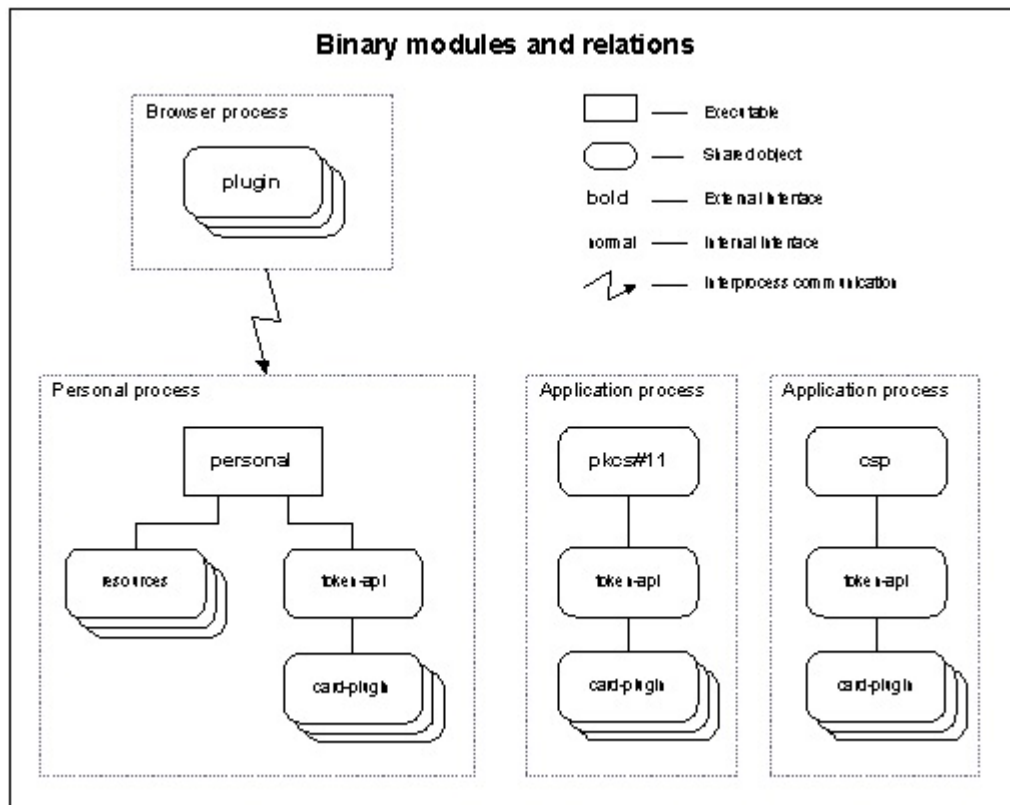
Browsers normally cache the session keys and this could result in a security flaw unless the session is terminated when the token is removed. As it is not possible to kill only the ongoing SSL session(s), the

browser itself is terminated causing all its current sessions to be terminated. NetDetacher is configured in the Personal GUI. See reference [4] for further information.

Product Structure

Personal is made up of a set of binary modules related either within a process or inter-process. An overview of the system is given in the following figure.

Figure 1. Product Structure



Personal Process

The Personal process, or Personal application, is the main executable in the client system. The application contains various administrative features and provides all online functions (signing, authentication, enrollment, etc.).

It contains the following components:

Personal	The main executable.
Resources	The collection of language and GUI resources contained in shared object modules, e.g. .dll, .so and .dylib.
Token-API	The internal API module dynamically loaded to provide token and cryptographic functions.
Card plug-ins	A number of shared objects implementing support for specific cards and tokens.

Browser Process

The plug-in function interface is implemented as both an ActiveX object (*Windows only*) and a NPAPI plug-in, loaded by the browser process, in order to support Internet Explorer and Mozilla-

based browsers. This object is thin and mainly implements communication with the main Personal process, where the actual online functions are implemented.

Third-Party Applications

In order to support Internet Explorer (*Windows only*) and Mozilla-based browsers that need to access the token or cryptographic functions during SSL handshake, Personal exposes the APIs through Personal PKCS#11 and Personal CSP (*Windows only*). Even other third-party applications make use of these APIs.

GUI Branding

The “standard” GUI of Personal can be replaced with a branded version. In a branded version of Personal it is possible to replace any dialogs, icons or text strings with other resources. This is done with branding modules. How the branding is done on various platforms is described in “Branding” on page 16. For further information about how to brand Personal please contact the Personal distributor.

Environment

See the `release.txt` file for information on supported platforms and web browsers in Personal.

What is New in this Version

For details on new or changed features in this version/release, see `Release.txt`.

Where to Find the Information

In addition to the introduction in this chapter, a more detailed description of the components in Personal is found in chapter “Functional Description Overview” on page 9.

Programmers and integrators will find descriptions of the various programming interfaces in the following chapters:

- “*WebSigner Plug-in*”
- “*Signer2 Plug-in*”
- “*Authentication Plug-in*”
- “*Registration Utility Plug-in*”
- “*Administration Plug-in*”
- “*LogoutTokens Plug-in*”
- “*Version Plug-in*”
- “*Personal PKCS#11*”
- “*Personal CSP*” (*Windows only*).

There are three separate chapters describing installation. Chapter “Installation on Windows” on page 103, chapter “Installation on Macintosh” on page 107 and chapter “Installation on Linux” on page 109 contains information about how the installation works and what options are available to OEM customers and integrators.

Chapter “Administration” on page 111 contains information about the functions available via the Personal GUI. The appendices present topics on various technical matters. Other sources of information are listed in “References” on page 6.

Release.txt

New functions and last minute information about Personal are described in the `release.txt` file.

Help

Personal is provided with a help file.

How to Contact Us

Development, maintenance, and support of Personal are managed by Technology Nexus AB .

To provide feedback about our products or to suggest product enhancements, please send an e-mail to `<contact@nexussafe.com>`.

Chapter 1. Functional Description Overview

Architecture

The plug-ins are designed to be as small and reusable as possible. To achieve this, every plug-in is implemented in a three-layered architecture. Instead of making the plug-ins perform specific operations, the main task is to pass on data to the Personal application, where plug-in handlers carry out the actual work.

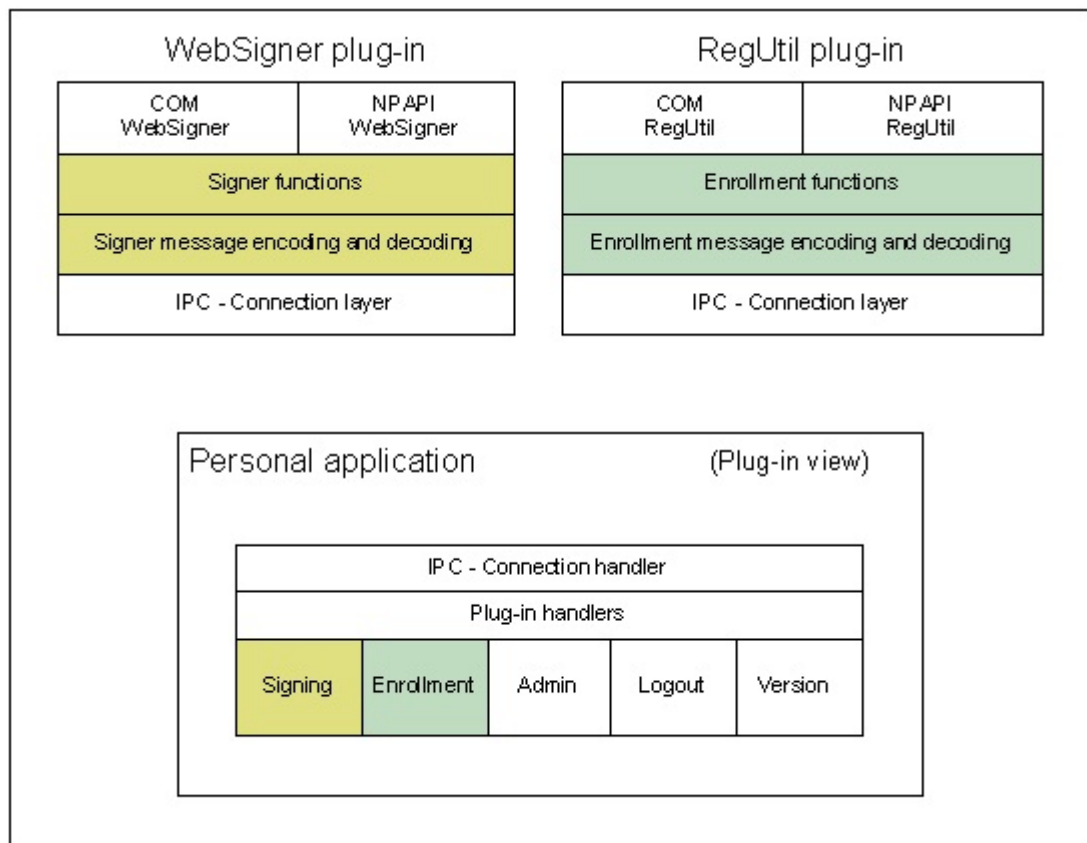
The three layers are:

- Browser specific (top)
- Plug-in specific (middle)
- Platform specific (bottom)

Figure “Plug-in Architecture ” shows an example of the layers in a signing and an enrollment component. The *top layer* is the actual browser specific interface, COM (*Windows only*) or NPAPI. The two tiers *middle layer* contains the plug-in specific implementation, minimal to only handle messages. The *bottom layer* is the platform specific inter-process communication implementation.

The actual signing functions are handled by the Signing plug-in handler in Personal application, and in the same way the enrollment functions are handled by the Enrollment plug-in handler.

Figure 1.1. Architecture



A plug-in communicates with the Personal application using messages sent via inter-process communication (IPC) calls.

On the Windows platform, the Personal application is implemented as a COM-server, and the plug-ins as COM-clients.

On the Mac OS X platform, an Apple script server is used.

On the Linux Ubuntu platform, a Unix named pipe is used.

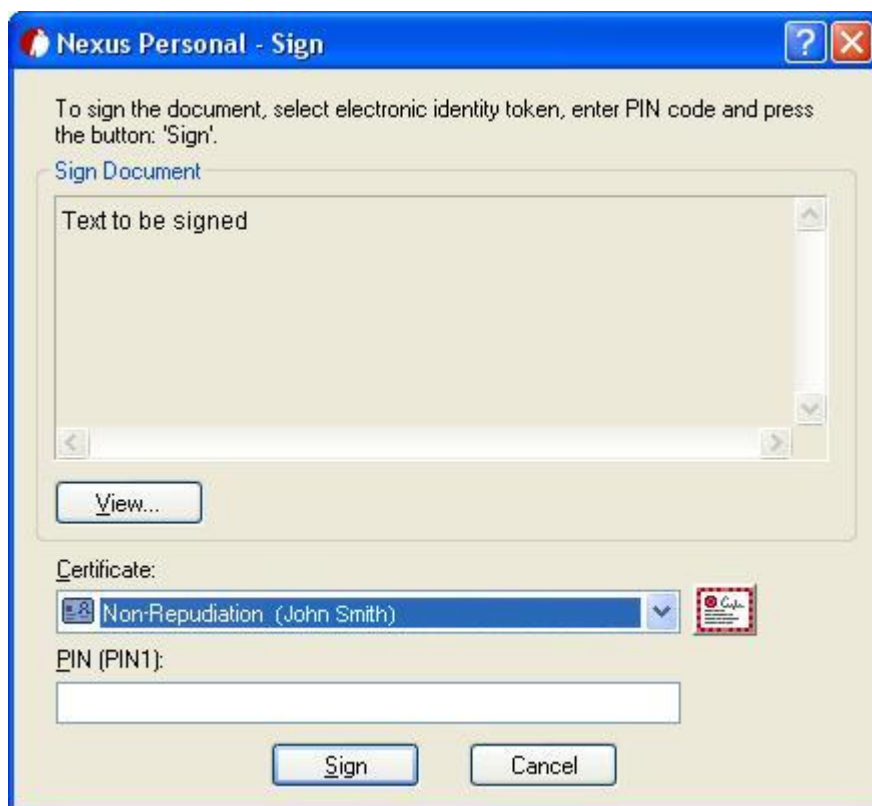
WebSigner

The WebSigner plug-in is used to create digitally signed messages in web browsers. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*), and as a plug-in for Mozilla-based browsers.

The user is prompted for his PIN code to enable access to the token. When access to the token is enabled, WebSigner will compute the signature and send a PKCS #7-SignedData message to the specified server application. The user will be notified by an error message if the PIN code is incorrect.

When WebSigner is activated to sign plain text, the data will be displayed in a signature window. Press the View button to see the data to be signed in a separate application window. The application, to which the MIME type of the data is associated will be launched, and the data to be signed can be printed or saved to file.

Figure 1.2. WebSigner



When WebSigner is activated to sign a file, the signature window is the same as when signing plain text. Again the data to be signed can be viewed, saved to file, or printed by pressing the View button. For more information, see chapter “ WebSigner Plug-in ”.

Signer2

The Signer2 plug-in is used to create digitally signed messages or files in webbrowsers. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*), and as a plug-in for Mozilla-based browsers.

When creating a digital signature, the user is prompted for his PIN code in order to enable access to the token. When access to the token is enabled, Signer2 will compute the signature in XML Digital Signature format. The resulting signature can be sent to a web server for verification.

The Signer2 plug-in is designed to prevent so called “Man-In-The-Middle” attacks. A DNS lookup is always performed on the URL from which the plug-in is called. The resulting IP address is included in the signature.

For more information, see chapter “ Signer2 Plug-in”.

Authentication

The Authentication plug-in can be used for application level authentication to web servers. It is an alternative to client side SSL authentication provided in web browsers. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*), and as a plug-in for Mozilla-based browsers.

When creating a digital signature for authentication, the user is prompted for his PIN code in order to enable access to the token. When access to the token is enabled, Authentication plug-in will compute the signature in XML Digital Signature format. The resulting signature can be sent to a web server for verification. If the signature is successfully verified, the user is authenticated to the web server.

The Authentication plug-in is designed to prevent so called “Man-In-The-Middle” attacks. A DNS lookup is always performed on the URL from which the plug-in is called. The resulting IP address is included in the signature. The Authentication plug-in provides a mechanism for token removal detection. It is possible to register an URL to which Personal should post a message if the user removes his/her token within a defined time frame. For more information, see chapter “ Authentication Plug-in”.

Registration Utility

The Registration Utility makes it possible for a user to connect to a Certification Authority, such as Nexus Certificate Manager, to enroll certificates and store them on a token. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*) and as a plug-in for Mozilla-based browsers.

If a software token or key pairs are not available, they will be created. In addition to token and certificate information, Registration Utility supports PIN policies to be set for soft tokens. These policies are enforced when the user is trying to change the PIN codes. In addition, one-time passwords can be used, which are sent to the Certification Authority for validation and authorization during the enrollment process.

The Registration Utility can be launched with a GUI, in which token name and PIN code are entered by the user, or in silent mode, i.e. without interaction with the user.

Figure 1.3. Registration

For more information see chapter “Registration Utility Plug-in”.

Administration Plug-in

The Administration plug-in is used to manage tokens and the token PINs via a web browser. The Administration plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*), and as a plug-in for Mozilla-based browsers.

The plug-in invokes the various wizards available in Personal. For more details, see the Help file in Personal.

Version Plug-in

In order to retrieve information about the installed components, there is a Version plug-in in Personal. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*) and as a plug-in for Mozilla-based browsers.

The Version plug-in returns the installed Personal components and their respective versions. The output from the Version plug-in is a formatted string with the details of the installed components. The version string can either be posted to a web application or returned by a method for further processing in a script.

For more information see chapter “Version Plug-in”.

LogoutTokens Plug-in

The LogoutTokens plug-in allows the web server to log out a user from the token once the session has been completed. The plug-in is implemented as an ActiveX control for Internet Explorer (*Windows only*) and as a plug-in for Mozilla-based browsers.

After the user has finished the session and logs out from the current web site, the web server invalidates the session key by clearing it at the server side. The log out from the server prevents someone else from creating a new session without entering the PIN again.

For more information see chapter “LogoutTokens Plug-in”.

Cryptographic APIs

Personal supports the cryptographic APIs PKCS #11 and Microsoft CryptoAPI (*Windows only*). Both APIs rely upon the cryptographic mechanisms and tokens provided by the Token API. Personal CSP and PersonalPKCS#11 are implemented in `personal.dll`.

Microsoft CSP

Note

This section applies to Windows only.

Personal includes a Microsoft CSP of type PROV_RSA_FULL. Hence, the tokens supported by Personal Token-API are exposed in the Microsoft CryptoAPI. The certificates in Personal are imported to, and removed from, Microsoft Certificate Store by a process continuously running in the `personal.exe` application.

Whenever necessary, the CSP will present a PIN dialog window, for input of the PIN code needed to access the private keys in the token. This will happen, e.g. when accessing a secure web server that requests client authentication.

Figure 1.4. Authentication



For more information see chapter “Personal CSP”.

PKCS#11

Personal includes a PKCS#11 compliant library. By using this library, third party applications can interact with the cryptographic functions and tokens in the underlying Personal Token-API.

For more information see chapter “Personal PKCS#11”.

Installation and Updating

On Windows, Personal is installed and updated over the web. In order to achieve this, Personal installation file is contained in a signed CAB file used for downloading and installing through Internet Explorer, and an XPI file for Mozilla-based browser installations. These files are located on the web server, and when a web browser accesses the site, Personal is installed if it is not already present on the client.

By calling the Version plug-in from a web application script, the installed version will be detected, and if it is outdated, the web browser can be redirected to a web site where an updated CAB or XPI file is located.

For traditional installation scenarios and for web browsers that do not allow CAB/XPI files to be executed, the `personalsetup.exe` file is also provided.

The Personal installation program is a dedicated application, called `persinst.exe`, containing all the necessary files and the information needed to install them.

For more information see chapter “Installation on Windows”.

On Mac OS X, the application name is Personal.app. When the user starts Personal for the first time, an installation of the different components takes place. For more information see chapter “Installation on Macintosh”.

On Linux, Personal is delivered as a compressed tar file. To install Personal the user unpacks the tar file and runs an installation script present in the package. When Personal is installed it is accessible from the Applications menu on the desktop. For more information see chapter “Installation on Linux”.

Soft Token Migration

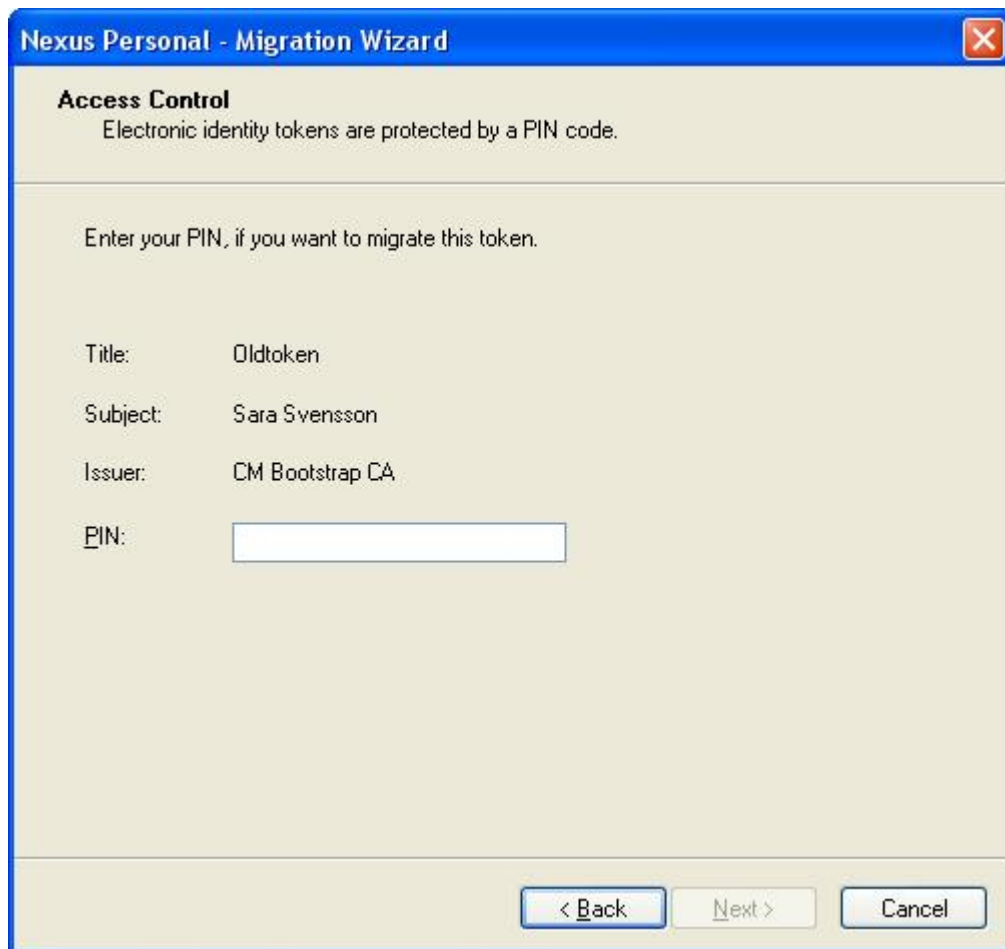
To allow import of soft tokens configured for iD2 Personal 2.x and SmartTrust Personal 3.x , soft token migration is available (*Windows only*).

Migration of tokens on the Macintosh platform is available from Personal 3.5 and 3.6 (*MAC OS X only*).

The migration process is carried out by a wizard. The wizard is either launched automatically the first time Personal is used or any other time from the Migrate command in the File menu.

The soft tokens to be migrated are identified by the mount points in iD2 Personal 2.x and SmartTrust Personal 3.x configuration files. The user gets the possibility to select the tokens to migrate and to enter the PIN codes for these tokens.

Figure 1.5. Soft Token Migration



When the soft tokens are stored on the same drive as Personal is installed, the tokens are deleted upon completed migration. When the soft tokens are stored on another drive, e.g. on a floppy, the soft token remains on that disk.

Integration with Standard Products

Personal is a software that integrates with various standard products.

Integration with Internet Explorer

Note

This section applies to the Windows platform only.

Personal includes a CSP, which adds token support to Microsoft Internet Explorer. This integration automatically takes place during the installation.

Integration with Mozilla-Based Browsers

Personal includes a PKCS#11 module, which adds token support to Mozilla-based browsers. This integration automatically takes place during the installation.

Branding

The branding possibility is platform dependant.

Branding on Windows platforms

On the Windows platforms, the following modules may be branded:

- A language neutral DLL, branding.dll, which contains all icons and bitmaps.
- Language dependant DLLs, which contain all language specific resources such as dialogs and text strings.

Branding on Mac OS X

Branding and localizing Personal on Mac OS X relies on the standard Mac OS X localization and branding features part of Mac OS X Bundles.

In Mac OS X branding and localization can be done on a binary version of the product and does not require recompilation of the product from it source code. Branding and localization of the Personal application is done in three specific steps.

1. Changing string resources for localized text in the User Interface.
2. Replacing images displayed in the User Interface.
3. Changing NIB files for moving or changing layout for components.

Branding on Linux

Branding of the user interface is not supported on Linux.

Card Reader Support

All smart card communication is done through PC/SC interface. The new standard PC/SC 2.01 Part 10 is supported. That means that PIN-Pad readers are supported through the PC/SC interface in a standardized way.

Chapter 2. WebSigner Plug-in

Introduction

The WebSigner plug-in is used to create digitally signed messages in web browsers.

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	6969E7D5-223A-4982-9B79-CC4FAC2D5E5E	(Windows only)
ProgID	Nexus.SignerCtl	(Windows only)
Activation MIME type	application/x-personal-signer	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag, supplying none, some or all parameters using the <PARAM> tag.

Example 2.1. Example of an ActiveX control activation

```
<OBJECT ID="Signer" CLASSID="CLSID:6969E7D5-223A-4982-9B79-CC4FAC2D5E5E">
  <PARAM NAME='CharacterEncoding' VALUE='UTF8'>
  <PARAM NAME='DataToBeSigned' VALUE='Sign%20this.'>
  <PARAM NAME='PostURL' VALUE='https://server.com'>
</OBJECT>
```

Due to the Eolas Patent, chapter “ Appendix A — Eolas Patent ” supplies more information on how to activate a plug-in. If the WebSigner plug-in is started using direct activation, we recommend solution 2, while solution 1 is preferred if the plug-in is scripted.

At this stage, it is not necessary to set any parameters as they can be set later using the script functions.

It is also possible for the web server to use scripting to silently detect if the plug-in is installed in the client.

Example 2.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.SignerCtl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

It is recommended that the <OBJECT> tag is used to create the object to be used for signing. It is possible to use the plug-in object created using `new ActiveXObject(Nexus.SignerCtl)` but this object will not be initialized correctly by Internet Explorer. In other words, the browser functions needed for the signing operation can not be used i.e. the plug-in will not be able to post the signature by itself. Furthermore, Internet Explorer will not be set as a parent window to the signing window.

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the <OBJECT> tag by supplying some or all parameters using the <PARAM> tag. It must be noted that it is done in a different way than for Internet Explorer. The ClassID is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 2.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag

```
<OBJECT id="signer" type="application/x-personal-signer">
  <PARAM NAME='CharacterEncoding' VALUE='UTF8'>
  <PARAM NAME='DataToBeSigned' VALUE='Sign%20this.'>
  <PARAM NAME='PostURL' VALUE='https://server.com'>
</OBJECT>
```

Scripting can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 2.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  if(navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalsigner"]) {
        if (navigator
            .mimeTypes["application/x-personal-signer"].enabledPlugin) {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

The following parameters are used in the WebSigner interface. They are case sensitive if nothing else is stated explicitly.

Parameter	Explanation
Mime-type	The only supported MIME type is <code>text/plain</code> of the data to be signed. This type effects which application program to start if the View button in the WebSigner window is clicked. MIME types were originally specified in RFC 1341 but improvements have been made in other documents like RFC 1521 and RFC

	1522. If this parameter is not present, it defaults to <code>text/plain</code> .
<i>CharacterEncoding</i>	Sets the character encoding of the data to be signed, if relevant for the chosen MIME type. The only supported character encodings are <code>UTF8</code> and <code>platform</code> , where <code>platform</code> is the platform's default character encoding. The parameter is optional and if no character encoding is given, default will be <code>platform</code> . For Mac OS X, <code>platform</code> will be interpreted as <code>ISO 8859-1</code> . <code>UTF8</code> is recommended.
<i>Format</i>	<p>Defines the format of the output data. Currently, only the <code>PKCS#7</code> signed-data content type format is supported. After creation, the signature will be URL-encoded for sending as a web form element. The value <code>PKCS7SIGNED</code> specifies this format.</p> <p>For full backwards compatibility with versions of Personal prior to 3.0, the seconds may be removed by adding <code>_NoSeconds</code> to the <code>Format</code> parameter: <code>PKCS7SIGNED_NoSeconds</code>, <code>PKCS7SIGNED_Attached_NoSeconds</code>, or <code>PKCS7SIGNED_Detached_NoSeconds</code>. We also recommend that you use <code>PKCS7</code>, however, "<code>PKCS#7</code>" can still be used for backwards compatibility. As an option, the signed data may be either included in the resulting signature, by specifying <code>PKCS7SIGNED_Attached</code>, or excluded, by specifying <code>PKCS7SIGNED_Detached</code>. The parameter is optional and if no format is given, default will be <code>PKCS7SIGNED_Attached</code>.</p>
<i>HashAlg</i>	Optional parameter specifying which hash algorithms to use in signatures. Possible values are <code>MD5</code> , <code>SHA1</code> , <code>SHA224</code> , <code>SHA256</code> , <code>SHA384</code> , <code>SHA512</code> , <code>RIPEMD128</code> , and <code>RIPEMD160</code> . Default is <code>SHA1</code> .
<i>Filename</i>	An optional file name to be used as default in the Save dialog when the user presses the Save... button in the WebSigner dialog box.
<i>WindowName</i>	If present, this parameter specifies the name of the window or frame used to display the server response to the HTML post from WebSigner. <code>WindowName</code> is optional. If omitted, it defaults to the window (or frame), that WebSigner was activated in, i.e. <code>_self</code> .
<i>DataToBeSigned</i>	Used as the data to be signed when the data is embedded into the HTML page. The data to be signed must be sent URL-encoded. WebSigner will decode the message, present it in the signature dialog box and sign the decoded message.
<i>PostURL</i>	Defines the URL to which WebSigner will post the signed data. If this parameter is not defined, WebSigner will sign the data and make it available for later retrieval using the <code>GetSignature</code> script function, but it will not post the signature.
<i>PostParams</i>	If <code>PostURL</code> is set then the URL-encoded string with parameters is posted back.
<i>SignReturnName</i>	Defines the name of the form field to contain the signature posted by WebSigner. <code>SignReturnName</code> is optional and if not present the default is <code>SignedData</code> .

<i>DataReturnName</i>	Defines the name of the form field to contain the unsigned data posted by WebSigner. The original data will be returned. DataReturnName is optional and if not present, WebSigner will not post the unsigned data.
<i>VersionReturnName</i>	Returns the version of Personal when posting the signature. See DataReturnName and SignReturnName.
<i>Issuers</i>	<p>Defines the filter criteria based on Issuers used to reduce the user's certificate choices when signing. Specific certificate attribute search strings can be specified, separated by "," or ";" where comma is interpreted as logical AND and semicolon as logical OR. The following X.500 attribute abbreviations are available: cn, g, s, t, ou, o, email, i, sn, street, l, st, c, d, and dc . In addition, OIDs can be used.</p> <p>Regular expressions using the wildcards * and ? can also be used. * matches an arbitrary number of characters. ? matches exactly one character. To match a string containing an asterisk or questionmark, the wildcard must be escaped using a backslash \. So to match an asterisk simply type *.</p> <p>Example 1: The search string cn=Our CA* will filter out all certificates issued by CAs with common name starting with Our CA.</p> <p>Example 2: 2.5.4.6=SE will filter out all Swedish certificates.</p>
<i>Subjects</i>	Defines the filter criteria based on Subjects used to reduce the user's certificate choices when signing. See Issuers.
<i>ViewData</i>	Switches between two default signature dialog boxes handled by WebSigner. The value <code>false</code> causes WebSigner to activate a small dialog box where the data to be signed is optionally viewed in a separate viewer application, which depends on the MIME type parameter. The value <code>true</code> causes WebSigner to activate a larger dialog box, which, in addition to the possibility to view the data separately, also displays the data in a text area within the dialog box.
<h3>Note</h3> <p>The data may be incorrectly displayed in the text area if it contains certain control characters such as the Null character. The default value is <code>true</code>.</p>	
<i>Base64</i>	If the signature is to be Base64-encoded before it is URL-encoded, this parameter must be set to <code>true</code> . Base64 is optional and case insensitive. If omitted or set to <code>false</code> , no Base64-encoding will be performed.
<i>IncludeCaCert</i>	Possible values are <code>true</code> and <code>false</code> (default is <code>false</code>). Will include the CA certificate chain (except the rootCA) in the signature, if available. IncludeCaCert is case insensitive.
<i>IncludeRootCaCert</i>	Possible values are <code>true</code> and <code>false</code> (default is <code>false</code>). Will include the Root CA certificate of the certificate chain in the signature, if available. IncludeRootCaCert is case insensitive.

UseBranding

An optional parameter that specifies if WebSigner should be branded or not. If the parameter is set to `true`, WebSigner GUI will be branded if Personal installation is branded. If the parameter is `false`, WebSigner GUI will not be branded, even if the installation is branded. If this parameter is not present, it defaults to `true`.

Scripting

The WebSigner profile may also be scripted using JavaScript or VB Script.

The following functions are available in Personal

Set<parameter name> `int Set<parameter name>(value);`

`String value;`

Parameters are set using the following:

The only exception is MIME type, which uses `SetMimeType`. The return value will always be 0.

Sign `int Sign();`

`int Sign()` pops up the signature dialog box and signs the data buffer. If `int Sign()` is successful, it returns 0 otherwise -1 is returned.

If the `PostURL` parameter is set, the plug-in will post the signature by itself. The signature can also be retrieved with `GetSignature()`, independent of it being posted. The retrieved signature can be posted using script.

GetVersion `String GetVersion();`

`String GetVersion()` returns the current WebSigner plug-in version number.

Note

To retrieve the Personal version, use the Version plug-in.

GetSignature `String GetSignature();`

If `Sign()` is successful (return 0) then the signature will be available by using the call `GetSignature()`. The signature is always URL-encoded. If `base64=true`, then base64 is used before being URL-encoded.

GetErrorString `String GetErrorString();`

If `Sign()` is not successful (return < 0) then an error string will be available by using the call `GetErrorString()`. The string is a null terminated ASCII string describing the error.

Usage and GUI

When WebSigner is activated, either through direct activation or by using the Sign() script function, it will display the signature dialog, allowing the user to see what he or she is about to sign. The user chooses the signing certificate and enters his or her PIN and clicks the Sign button.

Note

It is possible to filter the certificates made selectable for signing. See parameters Issuers and Subjects explained in “Parameters” and the examples in “ Sample Web Pages”.

If the data to be signed is plain text, it will be shown according to the appropriate character encoding using a fixed width font, if ViewData=true.

Branding of the WebSigner GUI is possible. See “ Branding”.

Sample Web Pages

This section includes some basic sample pages showing how to activate the plug-in. For more extensive samples, see the sample pages package.

Example 2.5. Example of Direct Activation using Internet Explorer (Windows only)

```
<HTML>
  <OBJECT ID="Signer" CLASSID="CLSID:6969E7D5-223A-4982-9B79-CC4FAC2D5E5E">
    <PARAM NAME='Mime-type' VALUE='text/plain'>
    <PARAM NAME='CharacterEncoding' VALUE='UTF8'>
    <PARAM NAME='Format' VALUE='PKCS7SIGNED_Attached'>
    <PARAM NAME='FileName' VALUE='text_tbs.txt'>
    <PARAM NAME='WindowName' VALUE='_self'>
    <PARAM NAME='DataToBeSigned' VALUE='Sign%20this:%20%C3%A5%C3%A4%C3%B6'>
    <PARAM NAME='PostURL' VALUE='http://server.com/post'>
    <PARAM NAME='PostParams' VALUE='userID=42&signID=4711'>
    <PARAM NAME='SignReturnName' VALUE='SignedData'>
    <PARAM NAME='VersionReturnName' VALUE='Version'>
    <PARAM NAME='Issuers' VALUE='cn=Our CA,c=SE;cn=Your CA,c=FI'>
    <PARAM NAME='Subjects' VALUE='cn=Test,c=SE'>
    <PARAM NAME='ViewData' VALUE='true'>
    <PARAM NAME='Base64' VALUE='false'>
    <PARAM NAME='IncludeCaCert' VALUE='true'>
    <PARAM NAME='IncludeRootCaCert' VALUE='false'>
  </OBJECT>
</HTML>
```

Example 2.6. Example of Scripting using Internet Explorer (*Windows only*)

```

<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.SignerCtl");
      if(xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>
  <OBJECT ID="signer" CLASSID="CLSID:6969E7D5-223A-4982-9B79-CC4FAC2D5E5E">
</OBJECT>
  <SCRIPT language="JavaScript">
    signer.SetMimeType('text/plain');
    signer.SetCharacterEncoding('platform');
    signer.SetFormat('PKCS7SIGNED_Attached');
    signer.SetFileName('text_tbs.txt');
    signer.SetWindowName('_self');
    signer.SetDataToBeSigned('Sign this. ');
    signer.SetSignReturnName('SignedData');
    signer.SetDataReturnName('UnsignedData');
    signer.SetVersionReturnName('Version');
    signer.SetIssuers('');
    signer.SetSubjects('');
    signer.SetViewData('true');
    signer.SetBase64('true');
    signer.SetIncludeCaCert('true');
    signer.SetIncludeRootCaCert('true');
    if (signer.Sign() == 0) {
      document.writeln(signer.GetSignature());
    } else {
      document.writeln(signer.GetErrorString());
    }
  </SCRIPT>
</HTML>

```

Example 2.7. Example of Direct Activation using Mozilla-based browsers

```

<HTML>
  <OBJECT ID="signer" type="application/x-personal-signer">
    <PARAM NAME='Mime-type' VALUE='text/plain'>
    <PARAM NAME='CharacterEncoding' VALUE='UTF8'>
    <PARAM NAME='Format' VALUE='PKCS7SIGNED_Attached'>
    <PARAM NAME='FileName' VALUE='text_tbs.txt'>
    <PARAM NAME='WindowName' VALUE='_self'>
    <PARAM NAME='DataToBeSigned' VALUE='Sign%20this:%20%C3%A5%C3%A4%C3%B6'>
    <PARAM NAME='PostURL' VALUE='http://server.com/post'>
    <PARAM NAME='PostParams' VALUE='userID=42&signID=4711'>
    <PARAM NAME='SignReturnName' VALUE='SignedData'>
    <PARAM NAME='VersionReturnName' VALUE='Version'>
    <PARAM NAME='Issuers' VALUE='cn=Our CA,c=SE;cn=Your CA,c=FI'>
    <PARAM NAME='Subjects' VALUE='cn=Test,c=SE'>
    <PARAM NAME='ViewData' VALUE='true'>
    <PARAM NAME='Base64' VALUE='false'>
    <PARAM NAME='IncludeCaCert' VALUE='true'>
    <PARAM NAME='IncludeRootCaCert' VALUE='false'>
  </OBJECT>
</HTML>

```

Example 2.8. Example of Scripting using Mozilla-based browsers

```

<HTML>
  <SCRIPT language="JavaScript">
    if(navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
            navigator.mimeTypes["application/x-personalsigner"]) {
          if (navigator
              .mimeTypes["application/x-personal-signer"].enabledPlugin) {
            document.writeln("Plugin installed");
          }
        }
      }
    }
  </SCRIPT>
  <OBJECT id="signerId" type="application/x-personal-signer" length=0 height=0>
  </OBJECT>
  <SCRIPT language="JavaScript">
    var signer = document.getElementById('signerId');
    signer.SetMimeType('text/plain');
    signer.SetCharacterEncoding('platform');
    signer.SetFormat('PKCS7SIGNED_Attached');
    signer.SetFileName('text_tbs.txt');
    signer.SetWindowName('_self');
    signer.SetDataToBeSigned('Sign this. ');
    signer.SetSignReturnName('SignedData');
    signer.SetDataReturnName('UnsignedData');
    signer.SetVersionReturnName('Version');
    signer.SetIssuers('cn=Our CA,c=SE;cn=Your CA,c=FI');
    signer.SetSubjects('cn=Test,c=SE');
    signer.SetViewData('true');
    signer.SetBase64('true');
    signer.SetIncludeCaCert('true');
    signer.SetIncludeRootCaCert('true');
    if (signer.Sign() == 0) {
      document.writeln(signer.GetSignature())
    } else {
      document.writeln(signer.GetErrorString());
    }
  </SCRIPT>
</HTML>

```

Digital Signature Format

This section defines the format of the returned signature. The signature is formatted according to "PKCS #7 v1.5 RSA Cryptographic Message Syntax Standard" and can be Base64 -encoded for transport.

The encrypted digest within the PKCS #7 object is encrypted according to RSAES-PKCS-v1_5 (Reference PKCS #1 v2.0).

The PKCS #7 (v1.5) object is a ContentInfo object with content of type SignedData identified by the signedData OID. The fields of the SignedData object have the following values:

Field	Value
<i>Version</i>	1
<i>digestAlgorithms</i>	SHA-1 object identifier
<i>contentInfo.contentType</i>	PKCS #7 Data object identifier
<i>contentInfo.content</i>	The signed text is included by default. The default behavior can be overridden by choosing a different value for the Format

parameter. [10] PKCS7SIGNED_Attached ensures that data is present, while PKCS7SIGNED_Detached ensures that data is not present in the signature. The parameter is optional and if no format is given, default will be PKCS7SIGNED_Attached.

Note

We recommend that you use PKCS7 but "PKCS#7" can still be used for backwards compatibility.

<i>certificates</i>	The signing certificate is included by default. If the parameters IncludeCaCert and IncludeRootCaCert are set to true, then the entire certificate chain up to the root is included (if found). Certificates may appear in any order.
<i>Crls</i>	Not present.
<i>SignerInfo.version</i>	1
<i>SignerInfo.issuerAndSerialNumber</i>	The issuer and serial number of the identity certificate.
<i>SignerInfo.digestAlgorithm</i>	SHA-1 object identifier
<i>SignerInfo.authenticatedAttributes</i>	Three attributes are present: A PKCS #9 content type attribute, the value of which is the same as SignData's contentInfo.contentType. In this case this is the PKCS #7 Data object identifier. A PKCS #9 message digest attribute, the value of which is the message digest of the content. A PKCS #9 signing time attribute, the value of which is the time at which the object was signed.

Note

For full backwards compatibility with versions of Personal prior to 3.0, the seconds may be removed by adding *_NoSeconds* to the Format parameter, (i.e. PKCS7SIGNED_NoSeconds, PKCS7SIGNED_Attached_NoSeconds, or PKCS7SIGNED_Detached_NoSeconds). We also recommend that you use PKCS7, however, "PKCS#7" can still be used for backwards compatibility.

<i>SignerInfo.digestEncryptionAlgorithm</i>	PKCS#1 rsaEncryption object identifier
<i>SignerInfo.encryptedDigest</i>	The result of encrypting the message digest (BERencoded DigestInfo) of the complete DER-encoding of the Attributes value contained in the authenticatedAttributes field with the signer's private key. See Reference PKCS#7, Section 9.3, with the clarifying footnote. The data is encrypted according to RSAESPKCS-v1_5 (Reference PKCS #1 v2.0).

Chapter 3. Signer2 Plug-in

Introduction

The Signer2 plug-in is used to digitally sign messages or files in web browsers. It creates an XML signature in accordance with the BankID specification (in reference [8]).

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	FB25B6FD-2119-4CEF-A915-A056184C565E	(Windows only)
ProgID	Nexus.SignerV2Ctl	(Windows only)
Activation MIME type	application/x-personal-signer2	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag. The parameters are set later using a scripting language.

Example 3.1. Example of an ActiveX control activation

```
<OBJECT ID="signer"
        CLASSID="CLSID:FB25B6FD-2119-4CEF-A915-A056184C565E">
</OBJECT>
```

It is also possible for the web server to use a scripting language to silently detect if the plug-in is installed in the client.

Example 3.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.SignerV2Ctl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the <OBJECT> tag. It must be noted that it is done in a different way than for Internet Explorer. The ClassID is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 3.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag

```
<OBJECT id="signer" type="application/x-personal-signer2">
</OBJECT>
```

Scripting can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 3.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  if (navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalsigner2"]) {
        if (navigator
            .mimeTypes["application/x-personal-signer2"].enabledPlugin) {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

This section describes which parameters are defined for the Signer2 plug-in. The parameters can be set by calling the function `SetParam` and retrieved by calling `GetParam`. To reset all parameters of the plug-in, call the function `Reset`.

The parameters *Issuers*, *Subjects* and *Policies* when combined use the following heuristic.

- If *Issuers* and *Subjects* are set, logical AND is used to filter the result.
- If *Issuers* and *Policies* are set, logical OR is used to filter the result.
- If *Subjects* and *Policies* are set, logical AND is used to filter the result.
- If all three are set, the following apply: (*Issuers* AND *Subjects*) OR *Policies*

They are case sensitive if nothing else is stated explicitly.

Parameter	Explanation
<i>Issuers</i>	<p>Defines the filter criteria based on Issuers used to reduce the user's certificate choices when doing a signing operation. Specific certificate attribute search strings can be specified, separated by , or ; where comma is interpreted as logical AND and semicolon as logical OR. The following X.500 attribute abbreviations are available: cn, g, s, t, ou, o, email, i, sn, street, l, st, c, d, and dc. In addition, OIDs can be used.</p> <p>Regular expressions using the wildcards '*' and '?' can also be used. * matches an arbitrary number of characters. ? matches exactly one character. To match a string containing an</p>

	<p>asterisk or questionmark, the wildcard must be escaped using a backslash \. So to match an asterisk simply type *.</p> <p>Example 1: The search string <code>cn=Our CA*</code> will filter out all certificates issued by CAs with common name starting with <code>Our CA</code>.</p> <p>Example 2: <code>2.5.4.6=SE</code> will filter out all Swedish certificates.</p>
<i>Subjects</i>	Defines the filter criteria based on Subjects used to reduce the user's certificate choices when doing a signing operation. See Issuers.
<i>Policys</i>	Defines the filter criteria based on Policys used to reduce the user's certificate choices when doing a signing operation. See Issuers. Note: Logical AND is not applicable for Policys. Regular expressions are not applicable for Policys.
<i>TextToBeSigned</i>	Mandatory. The text to be shown to the user. The value should be Base64-encoded. Character encoding of the text is defined by parameter <i>TextCharacterEncoding</i> .
<i>TextCharacterEncoding</i>	Optional. The character encoding of the (shown) text to be signed. Value could be <code>ISO-8859-1</code> or <code>UTF-8</code> . Default is <code>UTF-8</code> .
<i>Nonce</i>	Mandatory. Limited UTF-8 encoded string. The value should be Base64-encoded.
<i>ServerTime</i>	Optional. May contain a timestamp for the reference/tracability by the server. The value must specify the number of seconds from midnight January 1st 1970, UTC.
<i>NonVisibleData</i>	Optional. Size limited amount of data. Parameter value should be Base64-encoded. Max 5Mb (after Base64-encoding).
<i>RefDigestMethod</i>	Optional. Digest algorithm used when creating the references in the XML signature. The value may be <code>SHA1</code> or <code>SHA256</code> . Default is <code>SHA256</code> .
<i>SignMethod</i>	Optional. Encryption method for the signature. Today only <code>RSA-SHA1</code> is supported.
<i>OnlyAcceptMRU</i>	An optional parameter that specifies if only the last used token (in Authentication/Signer2 plug-ins) should be available for signing. If set, it overrides all other filter parameters. If this parameter is not present, it defaults to false.
<i>Signature</i>	Only available through the function <code>GetParam</code> . Returns the signature created. The result is Base64-encoded.
<i>Version</i>	Only available through the function <code>GetParam</code> . Returns current version of the Signer2 plug-in.
<i>SupportedFileTypes</i>	<p>Only available through the function <code>GetParam</code>. Returns the supported file types and a flag indicating if there is reader software installed that can be used to view the file.</p> <p>Example 1: <code>txt=1&pdf=0&</code> The example above is returned from a version of Personal that supports both <code>.txt</code> and <code>.pdf</code>, but there is no software installed for viewing <code>.pdf</code>.</p>

Example 2: `txt=1` This example is returned from a version of BISP that only supports .txt and software is installed for viewing files of that type.

FileContent

Mandatory when signing files. The parameter value should be Base64 encoded. The amount of data may not exceed 10MB or an error will be returned.

FileName

Mandatory when signing files. The parameter value should be Base64 encoded. The filename may not exceed 255 characters and should be UTF-8 before Base64 encoding.

Scripting

Following functions are exported from the signature interface

SetParam	<pre>int SetParam(paramType, paramValue);</pre> <pre>String paramType;</pre> <pre>String paramValue;</pre> <p>Returns Integer Errorcode.</p> <p>Function SetParam can be used to set parameters of the plug-in.</p>				
GetParam	<pre>string GetParam(paramType);</pre> <pre>String paramType;</pre> <p>Returns String paramValue.</p> <p>If an empty string is returned, the command has failed. The error code can be retrieved with a call to the function GetLastError.</p>				
PerformAction	<pre>int PerformAction(action);</pre> <pre>String action;</pre> <p>Returns Integer Errorcode.</p> <p>This function requires that the plug-in is loaded from an SSL protected web page.</p> <p>The following actions are yet available:</p> <table border="1"> <thead> <tr> <th>Action</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Sign</td> <td>Generate XML signature</td> </tr> </tbody> </table>	Action	Description	Sign	Generate XML signature
Action	Description				
Sign	Generate XML signature				
GetLastError	<pre>int GetLastError();</pre> <p>Returns Integer Errorcode.</p> <p>Call this function in order to retrieve the last error code of the plug-in. Useful when for example function GetParam returns NULL and one wants to know the reason of the error.</p>				
Reset	<pre>int Reset();</pre>				

Returns Integer Errorcode.

When called, all plug-in parameters are reset.

Sample Web Pages

This section includes some basic sample pages showing how to activate the plug-in.

Internet Explorer

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.SignerV2Ctl");
      if(xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>

  <OBJECT ID="signer2" CLASSID="CLSID:FB25B6FD-2119-4cef-A915-A056184C565E">
</OBJECT>

  <SCRIPT language="JavaScript">
    signer2.SetParam('TextToBeSigned', 'SGVsbG8h');
    signer2.SetParam('Nonce', 'OTU4ZTZmZWU=');
    signer2.SetParam('ServerTime', '1221630668');
    var res = signer2.PerformAction('Sign');
    if (res == 0) {
      document.writeln('Signature successfully created.');
```

Mozilla-based browsers

```
<HTML>
  <SCRIPT language="JavaScript">
    if(navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
            navigator.mimeTypes["application/x-personal-signer2"]) {
          if (navigator
              .mimeTypes["application/x-personal-signer2"].enabledPlugin) {
            document.writeln("Plugin installed.");
          }
        }
      }
    }
  </SCRIPT>

  <OBJECT id="signer2" type="application/x-personal-signer2" length=0 height=0>
</OBJECT>

  <SCRIPT language="JavaScript">
```

```

signer2.SetParam('TextToBeSigned', 'SGVsbG8h');
signer2.SetParam('Nonce', 'OTU4ZTZmZWU=');
signer2.SetParam('ServerTime', '1221630668');
var res = signer2.PerformAction('Sign');
if (res == 0) {
    document.writeln('Signature successfully created.');
```

Signer2 Signature Sample

This is an example of a Signer2 signature complying with the BankID specification (see reference [8]).

```

<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <Reference Type="http://www.bankid.com/signature/v1.0.0/types" URI="#bidSignedData">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256" />
      <DigestValue>iq6k45Unm0HhFhx3DglUizghPvpKYpee38cesvQutA=</DigestValue>
    </Reference>
    <Reference URI="#bidKeyInfo">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256" />
      <DigestValue>OZ5gSjy9pEC2qQzkOWFR0rsCdxAdJyx8ZU8HThx1bc=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
M4wLxZM6M8QZ41+J5/Wj/TJ40ws6kpMzmG0mfi7UN4jrRu5gwi4Hn8M1xVxP17jZ2bG5J+W5AT YSeCu
LV1shTLATdC2tPtYn86RToh64odSV2gEWNz5lSogkmf9vegVsAcNIgaf0Z7mMB99XIvsWkRYGeqbsBn
LdYJu7ThRZxLyI=
  </SignatureValue>
  <KeyInfo Id="bidKeyInfo">
    <X509Data>
      <X509Certificate>
MIICGjCCAWqgAwIBAgICSXkwDQYJKoZIhvcNAQEFBQAwLDELMAkGA1UEBhMCU0UxZjAMBGNVBAoTBU5
leHVzMQ0wCwYDVQQDEwRDYSxMB4XDTA4MDkyMzA3NTczMFoXDTEwMDkyMzA3NTczMFoGTXEMBUGA1
UEAxMOTWFsaW4gRXJpa3Nzb24wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALEgjj7nDZOR9yWGK
luFXLHiuoQB15r9/sYkdjbrTLgCkZDIQ30dcNzlm fBV0NcentgI1Hc+yPkKSVzJbwGSizQdF9Mxfxb
D/JfzPvWfknAVPm7Z+Q+uSWNKmXBXb6 P88ud6E1US5Hf80dUWhYnlr+kICwdfAPpiSB3yPJ4/1BYeX
AgMBAAGjRTBDMakGA1UdEw QCMAAwEQYDVR0OBAoECEfJowHp23mVMBMGA1UdIwQMMaQACEju76E+CH
FUMA4GALUdDwE B/wQEAWIGQDANBgkqhkiG9w0BAQUFAAOCAQEASwHNA8f2h2jfgfHFNzfb0ztfDgng
JT7k9I31xrn9mqmZXX+5kNKE7bFT1YT3Q373PtvPhModDvAClA2Y/C4Alwn0T4jURVJ/tJQfUibY6hI
4yt1prV/CaFxy/0EgKTPAhQEibIKHgFpIvpcOXQH251ifpqfKt+FAG+tVT2qFu/hy0xsi2PNWD/oyIL
vkjMDnkruMqtGnIc7ASEeM73sRI7qBYrEKF7OwHw6aRSnqaGPrbVTy14muW0woQa9Q6jTVZmmSTwTHS
P19edyws48EraPQ3oaQ/p0qBe2ynx6uexaMLK22HCNHIsoL+yGGpcfa3K+cidiS6sjlSBD+frk/Q==
      </X509Certificate>
    </X509Data>
  </KeyInfo>
</Object>
  <bankIdSignedData xmlns="http://www.bankid.com/signature/v1.0.0/types" Id="bidSignedData">
    <usrVisibleData charset="UTF-8" visible="wysiwyg">SGVsbG8h</usrVisibleData>
    <srvInfo>
      <nonce>NjY4MDMyM2I=</nonce>
      <serverTime>1222158113</serverTime>
    </srvInfo>
    <clientInfo>
      <funcId>Signing</funcId>
    </clientInfo>
  </bankIdSignedData>
</Signature>

```


Code	Description
8020	Function not permitted.
8021	Personal busy. (Mac OS X only)
8022	Plug-in cannot communicate with Personal
8102	PIN is blocked
8551	The file type is not supported
8552	The file type has no associated application for preview

Chapter 4. Authentication Plug-in

Introduction

The Authentication plug-in can be used for logging in to web servers. It is an alternative to client side SSL authentication provided by web browsers. It creates an XML signature in accordance with the BankID specification. (See reference [8].)

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	DD137900-E4D7-4b86-92CC-2E968F846047	(Windows only)
ProgID	Nexus.AuthenticationCtl	(Windows only)
Activation MIME type	application/x-personal-authentication	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag.

Example 4.1. Example of an ActiveX control activation

```
<OBJECT ID="authentication"
        CLASSID="CLSID:DD137900-E4D7-4b86-92CC2E968F846047">
</OBJECT>
```

It is also possible for the web server to use scripting to silently detect if the plug-in is installed in the client.

Example 4.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.AuthenticationCtl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the `<OBJECT>` tag. It must be noted that it is done in a different way than for Internet Explorer. The `ClassID` is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 4.3. Example of how to activate the Mozilla-based browser plug-in using the `<OBJECT>` tag

```
<OBJECT id="authentication"
       type="application/x-personal-authentication">
</OBJECT>
```

Scripting can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 4.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  if (navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalauthentication"]) {
        if (navigator.mimeTypes["application/x-personalauthentication"].enabledPlugin) {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

The following parameters are used in the Authentication interface. The parameters can be set by calling the function `SetParam` and retrieved by calling `GetParam`. To reset all parameters of the plug-in, call the function `Reset`.

The parameters *Issuers*, *Subjects* and *Policies* when combined use the following heuristic.

- If *Issuers* and *Subjects* are set, logical AND is used to filter the result.
- If *Issuers* and *Policies* are set, logical OR is used to filter the result.
- If *Subjects* and *Policies* are set, logical AND is used to filter the result.
- If all three are set, the following apply: (*Issuers* AND *Subjects*) OR *Policies*

They are case sensitive if nothing else is stated explicitly. Parameter

Parameter	Explanation
<i>Issuers</i>	Defines the filter criteria based on Issuers used to reduce the user's certificate choices when doing an authentication operation. Specific certificate attribute search strings can be specified, separated by <code>,</code> or <code>;</code> where comma is interpreted

as logical AND and semicolon as logical OR. The following X.500 attribute abbreviations are available: cn, g, s, t, ou, o, email, i, sn, street, l, st, c, d, and dc. In addition, OIDs can be used.

Regular expressions using the wildcards * and ? can also be used. * matches an arbitrary number of characters. ? matches exactly one character. To match a string containing an asterisk or questionmark, the wildcard must be escaped using a backslash \. So to match an asterisk simply type *.

Example 1: The search string cn=Our CA* will filter out all certificates issued by CAs with common name starting with Our CA.

Example 2: 2.5.4.6=SE will filter out all Swedish certificates.

Subjects

Defines the filter criteria based on Subjects used to reduce the user's certificate choices when doing an authentication operation. See Issuers.

Policys

Defines the filter criteria based on Policys used to reduce the user's certificate choices when doing an authentication operation. See Issuers.

Note

Logical AND is not applicable for Policys.

Regular expressions are not applicable for Policys.

TokenRemovedURL

Optional (mandatory if *TokenRemovedTimeout* is used). Specifies the URL to which the client should make a connect if the user removes the token used when authenticating.

This parameter can also be set *after* a successful authentication and have immediate effect for the token used for the authentication (If *TokenRemovedTimeout* is also set). This is only possible within a pre-defined time frame after the authentication. After that time frame, a new authentication is required in order to set this parameter.

TokenRemovedTimeout

Optional (mandatory if *TokenRemovedURL* is used). Specifies a time in minutes (1–720) after which the function for disconnected token should no longer be activated. This means that if the user disconnects his/her token after the number of minutes specified by this parameter, no reconnection to *TokenRemovedURL* will be made.

This parameter can also be set *after* a successful authentication and have immediate effect for the token used for the authentication (If *TokenRemovedURL* is also set). This is only possible within a predefined time frame after the authentication. After that time frame, a new authentication is required in order to set this parameter.

Challenge

Mandatory. Contains the value to be signed as part of the authentication. Limited UTF-8 string, which has to be Base64-encoded.

<i>ServerTime</i>	Optional. May contain a timestamp for the reference/tracability by the server. The value should specify the number of seconds from midnight January 1st 1970, UTC.
<i>RefDigestMethod</i>	Optional. Digest algorithm used when creating the references in the XML signature. The value may be <i>SHA1</i> or <i>SHA256</i> . Default is <i>SHA256</i> .
<i>SignMethod</i>	Optional. Encryption method for the signature. Today only RSA-SHA1 is supported.
<i>Signature</i>	Only available through the function <code>GetParam</code> . Returns the signature created. The result is Base64-encoded.

Scripting

Following functions are exported from the authentication interface

`SetParam`

```
int SetParam(paramType, paramValue);
```

String paramType;
String paramValue;

Returns Integer Errorcode.

Function `SetParam` can be used to set parameters of the plug-in.

`GetParam`

```
string GetParam(paramType);
```

String paramType ;

Returns String paramValue.

If an empty string is returned, the command has failed. The error code can be retrieved with a call to the function `GetLastError`.

`PerformAction`

```
int PerformAction(action);
```

String action;

Returns Integer Errorcode.

This function requires that the plug-in is loaded from an SSL protected web page.

The following actions are yet available:

Action	Description
<i>Authenticate</i>	Generate authentication signature
<i>UnregisterURL</i>	Remove token removal detection for a given TokenRemovedURL

`GetLastError`

```
int GetLastError();
```

Returns Integer Errorcode.

Call this function in order to retrieve the last error code of the plug-in. Useful when for example function GetParam returns NULL and one wants to know the reason of the error.

Reset

```
int Reset();
```

Reset returns Integer Errorcode.

When called, all plug-in parameters are reset.

Sample Web Pages

This section includes some basic sample pages showing how to activate the plug-in.

Internet Explorer

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.AuthenticationCtl");
      if(xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>

  <OBJECT ID="authenticate"
    CLASSID="CLSID:DD137900-E4D7-4b86-92CC2E968F846047">
  </OBJECT>

  <SCRIPT language="JavaScript">
    authenticate.SetParam('TokenRemovedURL',
      'aHR0cDovL3Rlc3Quc2VydmVyLmNvbS' +
      '9Mb2dpbj9hY3Rpb249cmVtb3ZlZCZpZD0x');
    authenticate.SetParam('TokenRemovedTimeout', '1');
    authenticate.SetParam('Challenge', 'YmRhNWlYMTc=');
    authenticate.SetParam('ServerTime', '1221629266');
    var res = authenticate.PerformAction('Authenticate');
    if (res == 0)
    {
      document.writeln('Authentication signature successfully created.');
```

Mozilla-based browsers

```
<SCRIPT language="JavaScript">
  if(navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
        navigator.mimeTypes["application/x-personalauthentication"]) {
        if (navigator
          .mimeTypes["application/x-personalauthentication"]
```

```

        .enabledPlugin) {
            document.writeln("Plugin installed.");
        }
    }
}
</SCRIPT>
<OBJECT id="authenticate" type="application/x-personal-authentication"
length=0 height=0>
</OBJECT>
<SCRIPT language="JavaScript">
    authenticate.SetParam('TokenRemovedURL',
        'aHR0cDovL3Rlc3Quc2VydmlvLnNvbS' +
        '9Mb2dpbj9hY3Rpb249cmVtb3ZlZCZpZD0x');
    authenticate.SetParam('TokenRemovedTimeout', '1');
    authenticate.SetParam('Challenge', 'YmRhNWlyMTc=');
    authenticate.SetParam('ServerTime', '1221629266');
    var res = authenticate.PerformAction('Authenticate');
    if (res == 0)
    {
        document.writeln('Authentication signature successfully created.');
```

Authentication Signature Sample

This is an example of an Authentication signature complying with the BankID specification (see reference [8]).

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
<SignedInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
</CanonicalizationMethod>
<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
</SignatureMethod>
<Reference Type="http://www.bankid.com/signature/v1.0.0/types" URI="#bidSignedData">
<Transforms>
<Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
</Transform>
</Transforms>
<DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"></DigestMethod>
<DigestValue>bH0dj4Rgn7wn395SP5nc52R71B4Cex3aKW1a1D+cGqQ=</DigestValue>
</Reference>
<Reference URI="#bidKeyInfo">
<Transforms>
<Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
</Transform>
</Transforms>
<DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"></DigestMethod>
<DigestValue>nKZsSsN0+98sQw72xjcp0S+pf7+Cnwrzgyb9rywjeBM=</DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>
hPkmF23MnW8fSBeCwhhiYAWHyVDnxv181v83BZmxZv/xywjSJOTB5OP9rKwCZdYDqMwFLwm3
zo23Yxmb9u5nQ3fgh2Y1DpZ5AqeQRfhGyqdXwiu0LWW58jenCFBYny8gciUeiidq+DRFhuiM7
9QcoQ3WQ39AYYCKb360fC6+Kw=
</SignatureValue>
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#" Id="bidKeyInfo">
<X509Data>
<X509Certificate>
MIICgJCCAWggAwIBAgICSXgwDQYJKoZIhvcNAQEFBQAwLDELMAkGA1UEBhMCU0UxZjAMB
gNVBAoTBU5leHVzMQ0wCwYDVQQDEwRDYSAxMB4XDTA4MDkyMzA3NTcyOVoXDTEwMDkyMz
A3NTcyOVowGTEuBGA1UEAxMOTWFsaW4gRXJpa3Nzb24wgZ8wDQYJKoZIhvcNAQEBBQAw
```

```

DgY0AMIGJAoGBAMrB5J5+fiA2LkG6ZhXd02A/ZEi3V5rPPalNUGmpJfUVWnDm+p9J6oAB
rPUHTKWr6wgv2BHbCkiGbpQ6Q2Q2MCzewXmy6eig3fZHN+i6rtGHvdNMh4eQeywDL4aLf
NjLGFriAoA0ectlnk/NzD0tluQf/p4xQ0wyhIXIuQ0MGXP9AgMBAAGjRTBDMakGALUdEw
QCMAAwEQYDVR0OBAoECE5hhMhi3FdjMBMGALUdIwQMMaQACEju76E+CHFUMA4GALUdDwE
B/wQEAwIFoDANBgkqhkiG9w0BAQUFAAOCAQEAb4d3yCTVE9f7HbzHZagHgZAZ7IdQFf/t
wp9MzMoVbHs/bgrt/WttGWS1Go5dVHu2veeX08sVaHceg47WzgZCCYbLa2V9jRScR45T
fyBcl7J6qHm8xY3dVFndbVlAYESSqJS0bYkD/Cm+mPpKM8Fkhd7+s2enPFMmhyHrtOde
S9lG84J7Ft2DSpCtBj4xjWhrrd+Vd2pcfHbPlWm8kCDyP+gZHR04xC+NIjdrM8vYlF3t+
gGRFhf4m1st0k+3GprLlrrurg2TcCA9jkS0FjU8xmRB05wSV/rDjfhrt3Xif/cfSN9Ys28
z0iD3nFeJZfu4Xvi2EBMSW0Od+y1gv9WJg==
</X509Certificate>
</X509Data>
</KeyInfo>
<Object>
<bankIdSignedData xmlns="http://www.bankid.com/signature/v1.0.0/types"
Id="bidSignedData">
<srvInfo>
<nonce>NjZiNmQ3MzE=</nonce>
<serverTime>1222156707</serverTime>
</srvInfo>
<clientInfo>
<funcId>Identification</funcId>
<host>
<fqdn>reagan.liljeholmen.nexus.se</fqdn>
<ip>10.75.28.85</ip>
</host>
<version>
UGVyc29uYWxfZXhlPTQuMTAuMC4zMzYwZXJzaW5zdF9leGU9NC4xMC4wLjMzJkN0ZXN
0X25nX2V4ZT0xLjAuMC4xJnRva2VuYXBPX2RsbD00LjEwLjAuMjkmcGVyc29uYWxfZG
xsPTQuMTAuMC4yOSZucF9wcnNubF9kbGw9NC4xMC4wLjMzJmXuZi9zdnNlX2RsbD00L
jEwLjAuMzMmY3Jkc2l1bV9kbGw9NC4xMC4wLjMzJmNyZHNldGVjX2RsbD00LjEwLjAu
MzMmY3JkcHJpc2l1bV9kbGw9NC4xMC4wLjMzJmZic19zdnNlX2RsbD0xLjQuMC45JmJyX2V
udV9kbGw9MS40LjAuOSZicmFuZGl1Zl9kbGw9MS40LjAuOSZDU1BfSU5TVEFMTEVEPEV
RSVUUmUGVyc29uYWw9NC4xMC4wLjMzJmJnBsYXRmb3JtPXdpcjMyJm9zX3ZlcnNpb249d
2lueHAmYmVzdF9iZWZvcuU9MTIyNDc0MDI1MyY=
</version>
<env>
<ai>
<uhi>qEYzPslw1l1e0YuJf08h+yOerhg=</uhi>
<utb>cr1</utb>
<rpr>qkgYiWZcilq+D5PWSRe5mRCSr9U=</rpr>
<gbvv>YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnh5ekFCQ0RFRkdISUpKS0xNtk9QUVJTUVFVWM
DEyMzQ1Njc4OQ==</gbvv>
</ai>
</env>
</clientInfo>
</bankIdSignedData>
</Object>
</Signature>

```

Error codes

Code	Description
General Return Codes	
0	OK
8001	General error
8002	Operation cancelled by user
8003	Memory error
8004	Invalid parameter
8005	Failed to decode request
8006	Failed to encode request
8007	Failed to convert to/from Unicode

Code	Description
8008	Operation not supported
8009	Token not present
8010	Failed to determine page URL
8011	Server not trusted
8012	Parameter is not Boolean
8013	Incorrect PIN
8014	Parameter value is not numeric
8015	SSL required
8016	All parameters required for running PerformAction not set.
8017	Parameter value should be Base64-encoded.
8018	Invalid parameter value.
8019	Plug-in may not be called with IP address.
8020	Function not permitted.
8021	Personal busy. (Mac OS X only)
8022	Plug-in cannot communicate with Personal
8102	PIN is blocked
Authentication plug-in specific Return Codes	
8501	Such TokenRemovedURL not registered. Can be returned when action UnregisterURL is called.
8502	The parameters TokenRemovedURL and TokenRemovedTimeout are set too late after the last successful authentication. A new authentication is required in order to set these parameters.

Chapter 5. Registration Utility Plug-in

Introduction

The Registration Utility makes it possible to allow a user to connect to a Certification Authority, such as Nexus Certificate Manager, to request certificates and to store them on a token. If a software token or key pairs is not available, they are created.

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	AC7AEFE1-E745-4D21-881C-D7B6F22275B8	(Windows only)
ProgID	Nexus.RegUtilCtl	(Windows only)
Activation MIME type	application/x-personal-regutil	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag. The parameters are set later using a scripting language.

Example 5.1. Example of an ActiveX control activation

```
<OBJECT ID="RegUtil"
        CLASSID="CLSID:AC7AEFE1-E745-4D21-881C-D7B6F22275B8">
</OBJECT>
```

It is also possible for the web server to use a scripting language to silently detect if the plug-in is installed in the client.

Example 5.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.RegUtilCtl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

This ActiveXObject should not be used for enrollment. Instead the <OBJECT> tag must be used, as described above, for the object to be initialized correctly.

The Registration Utility plug-in is scripted only; therefore, we recommend that Solution 1 is used. Refer to “Appendix A — Eolas Patent” for more information.

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the <OBJECT> tag. It should be noted that this is done in a different way than for Internet Explorer. The ClassID is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 5.3. Example of how to activate the Mozilla-based browsers plug-in using the <OBJECT> tag

```
<OBJECT ID="RegUtil" TYPE="application/x-personal-regutil">
</OBJECT>
```

A script can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 5.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  if(navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalregutil"]) {
        if (navigator
            .mimeTypes["application/x-personal-regutil"].enabledPlugin) {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

Parameters available to SetParam are as follows

Type	Parameter Value
<i>tokenName</i>	Name of token to be created. If the tokenName and PIN are not set, Personal will pop-up a dialog for the user to input the data. It is not possible to just use tokenName parameter without using the PIN parameter.
<i>tokenType</i>	Specifies token type to enrol to. The following token types are supported: pkcs12 and internalstore. Default is internalstore.
<i>pin</i>	PIN code protecting the keys in the token to be created. If the PIN and tokenName are not set, Personal will pop-up a dialog for the user to input the data. It is not possible to just use PIN parameter without using the tokenName parameter.

<i>keySize</i>	Key size specified in bits. For example, 1024 or 2048.
<i>keyUsage</i>	Comma separated list of key usage strings according to X.509. Available values are: digitalSignature, keyEncipherment, nonRepudiation, and contentCommitment. Example, "digitalSignature,keyEncipherment".
<i>oneTimePassword</i>	One time password that will be verified on the server.
<i>subjectDN</i>	User's Distinguished Name in the certificate request. The following X.500 attribute abbreviations are available: CN, N, G, S, T, OU, O, DMD, SN, UI, STREET, L, ST, C, EM, D, and OID. Example "CN=Bob,C=SE" or "OID.2.5.4.3=Kalle".
<i>hashAlg</i>	Optional hash algorithm parameter to be used when signing request. Available values are MD5, SHA1, SHA224, SHA256, SHA384, SHA512, RIPEMD128, and RIPEMD160. Default is SHA1.
<i>maxLen</i>	Maximum allowed PIN length.
<i>minLen</i>	Minimum allowed PIN length.
<i>minChars</i>	Minimum number of characters that should be entered in a PIN.
<i>minDigits</i>	Minimum number of digits that should be entered in a PIN.
<i>rfc2797cmcoid</i>	An optional parameter. If set to true, it uses PKIData OID defined in RFC2797. The OID will be <code>id-cct-PKIData ::= {1.3.6.1.5.5.7.12.2}</code> . Default is false.

Scripting

The following functions are implemented

SetParam	<pre>int SetParam(paramType, paramValue);</pre> <pre>String paramType;</pre> <pre>String paramValue;</pre> <p>Returns Integer Errorcode.</p> <p>Sets a parameter.</p>
GetParam	<pre>String GetParam(paramType);</pre> <pre>String paramType;</pre> <p>Returns String paramValue.</p> <p>If an empty string is returned, the command has failed. The error code can be retrieved with <code>GetLastError()</code>.</p>
ValidatePin	<pre>int ValidatePin(PIN);</pre>

String *PIN*;

Returns Integer Errorcode.

The PIN is checked according to the PIN policy previously set. The policy is checked against the following parameters:

- *minLen*
- *minChars*
- *maxLen*
- *minDigits*
- *compareStr*
- *maxCompareInRow*
- *maxEqualInRow*

If no PIN policy has been specified, any PINs will be accepted. This is the default policy.

Note

The method remains for backwards compatibility. We recommend that the application handles the token name and PIN input data.

InitRequest

int **InitRequest**(*requestType*);

String *requestType*;

Returns Integer Errorcode.

InitRequest tells the plug-in that a request of the given type should be created when CreateRequest is called. It stores the given parameters, so they are available for the actual creation.

The value of *requestType* can be either *pkcs10* or *cmc* (default if not provided is “*cmc*”). To generate a PKCS #10 request, the parameters that must be set using SetParam are as follows:

- *keySize*
- *keyUsage*
- *subjectDN*

In addition, *hashAlg* can be used optionally. To generate a CMC request, the parameters that must be set are as follows:

- *oneTimePassword*

In addition, InitRequest with *requestType* equal to *pkcs10* must have been called twice before initializing the creation of the CMC request.

CreateRequest

string **CreateRequest**();

Returns String Base64-encoded certificate request.

This function creates the request previously specified by `InitRequest`. If the request previously specified is a CMC request, it also creates the two PKCS #10 requests as specified before and puts them into the CMC request.

Internally a new token is created, with one key pair for each PKCS #10 request that has been specified. If the `tokenName` and `PIN` parameters have not been set, Personal will present the user with a dialog, allowing him or her to enter a token name, and set a PIN for the token. The user must then enter a PIN which matches the specified PIN policy. If another token with the same name already exists, a number will be appended to the name in order to create a unique name.

If the `tokenName` and `PIN` parameters have been set, no dialog will be presented. If the PIN does not match the PIN policy, the function will fail. If a token with the same name already exists, a unique name will be created in the same way as specified above.

If an empty string is returned, the command has failed. The error code can be retrieved with `GetLastError()`.

`StoreCertificates`

```
int StoreCertificates(type,
certificateBlob);
```

```
String type;
String certificateBlob;
```

Returns Integer Errorcode.

The `type` must be `p7c` for a PKCS #7 certificate blob. The certificate BLOB should be a Base64-encoded response to the certificate request. The certificates will be stored in the appropriate token containing the public key(s) associated with the issued certificates. If no matching public key is found, it will not be possible to store the certificates. This means that if a CA certificate should be stored in a token, it must be bundled together with the user's certificates (which always should be the case).

Note

Personal does not support storing of Root CA certificates.

`GetLastError`

```
int GetLastError();
```

Returns Integer Errorcode.

`GetResponse`

```
string GetResponse(challenge);
```

```
String challenge;
```

Returns String.

GetResponse is called with a Base64 encoded challenge string to do client verification. Personal calculates the response and returns this as a Base64 encoded string. If an error occurred an empty string will be returned.

Reset

```
int Reset ();
```

Returns Integer Errorcode.

Resets all parameters.

Usage and GUI

Creating a Token

The web page sets the necessary parameters and calls InitRequest with request type set to `pkcs10`, to tell the plug-in that a PKCS #10 request should be created using the given parameter set.

The web page script might then change some parameters, for instance, `keyUsage`, and call InitRequest again, to tell the plug-in that a second PKCS #10 request should be created. If this is done, InitRequest must also be called a third time, but this time with type `cmc`, to specify that the two PKCS #10 requests should be packaged in a CMC request.

When the CreateRequest function is called, the plug-in will ask the Personal application to create a new token with the necessary key pairs and the specified requests. Please note that before this no communication between the plug-in and the Personal application had taken place. If the latest call to InitRequest specifies a PKCS #10 request, a token containing one key pair will be created, and one PKCS #10 request will be returned.

If the latest call to InitRequest specifies a CMC request, a new token containing two key pairs will be created and a CMC request containing two PKCS #10 requests will be returned.

If the `tokenName` and `PIN` parameters have not been set, Personal will present the user with a dialog, allowing him or her to enter a token name, and set a PIN for the token. The user must then enter a PIN which matches the specified PIN policy. If another token with the same name already exists, a number will be appended to the name in order to create a unique name.

If the `tokenName` and `PIN` parameters have been set, no dialog will be presented. If the PIN does not match the PIN policy, the function will fail. If a token with the same name already exists, a unique name will be created in the same way as specified above.

If the web page has set a PIN policy, using the available parameters, this will be stored in the token, and applied to the PIN that the user has set.

It is also possible to use the `ValidatePIN` method to verify a PIN against the specified PIN policy. However, this is not recommended as it is better for the application to handle the PIN input instead.

Store Certificates

The application will search for the token containing the same public keys as the ones in the certificates, and choose that token for storage of the certificates. Thus, no token name has to be entered at this stage.

If no matching public key is found, it will not be possible to store the certificates. This means that if a CA certificate should be stored in a token, it must be bundled together with the user certificates which always should be the case.

PIN Policy

If no PIN policy is set, the minimum length of the PIN will be set to 1 and the maximum to 255 characters. There will be no restrictions on the PIN.

If the web page specifies a PIN policy, this policy will be stored in the created token. This means that if the user changes the PIN, he or she will still have to follow the PIN policy.

However, this is only true as long as the token is kept in the Internal Store. If the token is exported to a PKCS #12 file, the PIN policy will be lost, since the PKCS #12 standard does not include any PIN policies. If the user then imports the PKCS #12 file into a Personal Internal Store again, the newly created token will not have any PIN policy restrictions.

Sample Web Pages

This section includes some basic sample pages, showing how to activate the plug-in.

Internet Explorer

Example 5.5. Example of creating a certificate request (*Windows only*)

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.RegUtilCtl");
      if(xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>

  <OBJECT ID="regutil"
    CLASSID="CLSID:AC7AEFE1-E745-4D21-881C-D7B6F22275B8">
  </OBJECT>

  <SCRIPT language="JavaScript">
    regutil.SetParam('keySize', '2048');
    regutil.SetParam('keyUsage', 'digitalSignature,keyEncipherment');
    regutil.SetParam('subjectDN', 'CN=BOB,O=Nexus,C=SE');
    regutil.InitRequest('pkcs10');
    regutil.SetParam('keyUsage', 'nonRepudiation');
    regutil.InitRequest('pkcs10');
    regutil.SetParam('oneTimePassword', '1234');
    regutil.SetParam('maxLen', '16');
    regutil.SetParam('minLen', '4');
    regutil.SetParam('minChars', '1');
    regutil.SetParam('minDigits', '1');
    regutil.SetParam('compareStr', "qwerty");
    regutil.SetParam('maxCompareInRow', '3');
    regutil.SetParam('maxEqualInRow', '2');
    regutil.InitRequest('cmc');
    document.writeln(regutil.CreateRequest());
    document.writeln(regutil.GetLastError());
  </SCRIPT>
</HTML>
```

Example 5.6. Example of storing the certificate response (Windows only)

```

<HTML>
  <OBJECT ID="regutil"
    CLASSID="CLSID:AC7AEFE1-E745-4D21-881C-D7B6F22275B8">
  </OBJECT>
  <SCRIPT language="JavaScript">
    regutil.StoreCertificates('p7c', <base64 encoded PKCS#7 certificate blob>);
  </SCRIPT>
</HTML>

```

Mozilla-based browsers

Example 5.7. Example of creating a certificate request

```

<HTML>
  <SCRIPT language="JavaScript">
    if(navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalregutil"]) {
          if (navigator.mimeTypes["application/x-personal-regutil"].enabledPlugin)
            {
              document.writeln("Plugin installed");
            }
        }
      }
    }
  </SCRIPT>

  <OBJECT ID="regutilId"
    TYPE="application/x-personal-regutil">
  </OBJECT>

  <SCRIPT language="JavaScript">
    var regutil = document.getElementById('regutilId');
    regutil.SetParam('keySize', '2048');
    regutil.SetParam('keyUsage', 'digitalSignature,keyEncipherment');
    regutil.SetParam('subjectDN', 'CN=BOB,O=Nexus,C=SE');
    regutil.InitRequest('pkcs10');
    regutil.SetParam('keyUsage', 'nonRepudiation');
    regutil.InitRequest('pkcs10');
    regutil.SetParam('oneTimePassword', '1234');
    regutil.SetParam('maxLen', '16');
    regutil.SetParam('minLen', '4');
    regutil.SetParam('minChars', '1');
    regutil.SetParam('minDigits', '1');
    regutil.SetParam('compareStr', "qwerty");
    regutil.SetParam('maxCompareInRow', '3');
    regutil.SetParam('maxEqualInRow', '2');
    regutil.InitRequest('cmc');
    document.writeln(regutil.CreateRequest());
    document.writeln(regutil.GetLastError());
  </SCRIPT>
</HTML>

```

Example 5.8. Example of storing the certificate response

```

<HTML>
  <OBJECT ID="regutilId"
    TYPE="application/x-personal-regutil">
  </OBJECT>
  <SCRIPT language="JavaScript">
    var regutil = document.getElementById('regutilId');
    regutil.StoreCertificates('p7c', <base64 encoded PKCS#7 certificate blob>);
  </SCRIPT>
</HTML>

```

Error Codes

Code	Description
0	OK
1	The call to StoreCertificate failed as a certificate could not be stored. The public key does not match that of any existing token.
630	PIN is too long or too short
631	Too few letters in the PIN
632	Too few digits in the PIN
633	Too many equal characters in a row
634	Too many characters from CompareStr exist in a row
640	Invalid parameter
641	Memory error
662	Failed to convert to/from UNICODE
666	Failed to convert to/from Base64
668	Failed to encode the request
669	Failed to decode the request
671	Failed to write certificates
672	subjectDn parameter has incorrect syntax
999	General Error
1024	Key length is not specified
1025	Key usage is not specified
1026	Invalid key usage
1027	PIN policy specification is inconsistent (e.g. minLen = 5 and maxLen = 4)
1028	Parameter is not numeric
1029	SetParam(HashAlg) was called with unknown algorithm name
1030	CreateRequest called without previous call to InitRequest
1031	An unsupported key length of less than 368 bits have used
1032	Parameter is not boolean.

Code	Description
1033	Token type is invalid.
1034	Operation cancelled by the user.
1035	Duplicate token name.
1036	Plug-in cannot communicate with Personal

Format of a CMC Request and Response

For detailed information about CMC refer to [5].

Request

The new content object PKIData is used as the body of the full PKI request message. PKIData can be defined in two ways.

Default is

```
id-ct-PKIData ::= { 1.3.6.1.5.5.7.5.2 }
```

When `rfc2797cmcoid = true` then

```
id-ct-PKIData ::= { 1.3.6.1.5.5.7.12.2 }
```

The ASN.1 contents is as follows:

- *controlSequence* will be empty.
- *reqSequence* consists of a sequence of PKCS#10 requests.
- *cmsSequence* will be empty.
- *otherMsgSequence* will contain the OID described in “One Time Password”.

One Time Password

The One Time Password (OTP) is stored in the *otherMsgSequence* field. It is identified by:

```
id-pin-code OBJECT IDENTIFIER ::= { 1.2.752.36.4.1.1
}
```

and is defined by:

```
PinCode ::= IA5String
```

Response

The certificates are always returned as a simple PKI response, i.e. a PKCS#7 signedData object.

Chapter 6. Administration Plug-in

Introduction

The Administration plug-in is used to manage tokens in Personal. It makes it possible to export, import and delete tokens as well as to administrate PINs. In the branding module (see “ Branding” , it is possible to configure the plug-in to only be run from a specific host.

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	524B98BC-7B94-48CB-8F6E-CEC7D1B64522	(Windows only)
ProgID	Nexus.WebAdminCtl	(Windows only)
Activation MIME type	application/x-personal-webadmin	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag. The parameters are set later using a scripting language.

Example 6.1. Example of an ActiveX control activation

```
<OBJECT ID="webadmin"
        CLASSID="CLSID:524B98BC-7B94-48CB-8F6E-CEC7D1B64522">
</OBJECT>
```

It is also possible for the web server to use a scripting language to silently detect if the plug-in is installed in the client.

Example 6.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
try {
  var xObj = new ActiveXObject("Nexus.WebAdminCtl");
  if(xObj) {
    document.writeln("Object installed.");
  }
} catch (e) {
  document.writeln("Object not installed.");
}
</SCRIPT>
```


This ActiveXObject should not be used for administration. Instead the <OBJECT> tag must be used, as described above, for the object to be initialized correctly.

The Administration plug-in is scripted only; therefore, we recommend that Solution 1 is used. Refer to “Appendix A — Eolas Patent”.

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the <OBJECT> tag by supplying some or all parameters using the <PARAM> tag. It must be noted that it is done in a different way than for Internet Explorer. The ClassID is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 6.3. Example of how to activate the Mozilla-based browser plug-in using the <OBJECT> tag

```
<OBJECT id="webadmin"
        type="application/x-personal-webadmin">
</OBJECT>
```

Scripting can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 6.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
if (navigator.plugins) {
  if (navigator.plugins.length > 0) {
    if (navigator.mimeTypes &&
        navigator.mimeTypes["application/x-personalwebadmin"]) {
      if (navigator.mimeTypes["application/x-personalwebadmin"].enabledPlugin) {
        document.writeln("Plugin installed");
      }
    }
  }
}
</SCRIPT>
```

Parameters

This section describes which parameters are defined for the Administration plug-in. The parameters can be set by calling the function SetParam. To reset all parameters of the plug-in, call the function Reset.

The parameters are case sensitive if nothing else is stated explicitly.

Parameter	Description
<i>Issuers</i>	Defines the filter criteria based on Issuers used to reduce the user's certificate choices when doing an administrative operation. Specific certificate attribute search strings can be specified, separated by "," or ";" where comma is interpreted as logical AND and semicolon as logical OR. The following X.500 attribute abbreviations are available: cn, g, s, t, ou,

o, email, i, sn, street, l, st, c, d, and dc. In addition, OIDs can be used.

Regular expressions using the wildcards * and ? can also be used. * matches an arbitrary number of characters. ? matches exactly one character. To match a string containing an asterisk or questionmark, the wildcard must be escaped using a backslash \. So to match an asterisk simply type *.

Example 1: The search string cn=Our CA* will filter out all certificates issued by CAs with common name starting with Our CA.

Example 2: 2.5.4.6=SE will filter out all Swedish certificates.

<i>Subjects</i>	Defines the filter criteria based on Subjects used to reduce the user's certificate choices when doing an administrative operation. See Issuers.
<i>PinOperation</i>	Optional parameter that specifies which kind of PIN operation should be performed. Following values are supported:
changePIN	Change PIN operation will be performed.
unlockPIN	Unlock PIN operation will be performed. If the parameter is not set the PIN administration operation will be initialized so that both change and unlock can be performed.
<i>challenge</i>	Base64 encoded challenge string. Used to set the challenge before a call to PerformAction with parameter getResponse is called.
<i>bbdInfo</i>	Base64 encoded xml-data that is used when a following call to PerformAction with parameter renewPollDates is called.

Scripting

All parameters should be in UTF-8 format in order for the plug-in to be able to treat the input data in a correct way.

Following functions are exported from the administration interface

```
SetParam          int SetParam(paramType, paramValue);
```

```
String paramType;
String paramValue;
```

Returns Integer Errorcode.

Function SetParam can be used to set parameters of the plug-in.

```
GetParam          string GetParam(paramType);
```

```
String paramType;
```

Returns String paramValue.

If an empty string is returned, the command has failed. The error code can be retrieved with a call to the function `GetLastError`.

`PerformAction`

```
int PerformAction(action);
```

```
String action;
```

Returns Integer Errorcode.

This function is called in order to initiate a specific administration action.

The following actions are yet available:

Action	Description
<code>pinAdministration</code>	Administrate PIN of a token. The operation is defined by the parameter <code>PinOperation</code> . If <code>PinOperation</code> is not set, both change and unblock PIN is allowed.
<code>exportToken</code>	Export a token from Personal. The token will be exported in the format given in the parameter <code>ExportType</code> . If <code>ExportType</code> is not set, the behaviour configured in the Personal installation will be used.
<code>importToken</code>	Import a token to Personal.
<code>deleteToken</code>	Delete a token from Personal.
<code>getResponse</code>	Calculates the response from a previous set challenge using <code>SetParam</code> .
<code>renewPollDates</code>	Requires that SSL is used. Sets Auto Update parameters from previous set data using parameter <code>bbdInfo</code> in <code>SetParam</code> . In the branding, it is also possible to define which servers are allowed to run the plug-in.

`GetLastError`

```
int GetLastError();
```

Returns Integer Errorcode.

Call this function in order to retrieve the last error code of the plug-in. Useful when for example function

`GetParam`

returns NULL and one wants to know the reason of the error.

`Reset`

```
int Reset();
```

Returns Integer Errorcode.

When called, all plug-in parameters are reset.

Sample Web Pages

This section includes some basic sample pages, showing how to activate the plug-in.

Internet Explorer

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.WebAdminCtl");
      if (xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>

  <OBJECT ID="webadmin"
    CLASSID="CLSID:524B98BC-7B94-48CB-8F6E-CEC7D1B64522">
  </OBJECT>

  <SCRIPT language="JavaScript">
    webadmin.SetParam('PinOperation', 'changePIN');
    webadmin.SetParam('Issuers', 'cn=Our CA,c=SE;cn=Your CA,c=FI');
    var res = webadmin.PerformAction('pinAdministration');
    if (res == 0) {
      document.writeln('Operation successfully performed. ');
    } else {
      document.writeln('Failed to perform action. Error = '+res);
    }
  </SCRIPT>
</HTML>
```

Mozilla-based browsers

```
<HTML>
  <SCRIPT language="JavaScript">
    if (navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
            navigator.mimeTypes["application/x-personalwebadmin"]) {
          if (navigator.mimeTypes["application/x-personal-webadmin"].enabledPlugin) {
            document.writeln("Plugin installed.");
          }
        }
      }
    }
  </SCRIPT>

  <OBJECT id="webadminId"
    type="application/x-personal-webadmin"
    length=0
    height=0>
```

```

</OBJECT>

<SCRIPT language="JavaScript">
var webadmin = document.getElementById('webadminId');
webadmin.SetParam('PinOperation', 'changePIN');
webadmin.SetParam('Issuers', 'cn=Our CA,c=SE;cn=Your CA,c=FI');
var res = webadmin.PerformAction('pinAdministration');
if (res == 0) {
    document.writeln('Operation successfully performed. ');
} else {
    document.writeln('Failed to perform action. Error = '+res);
}
</SCRIPT>
</HTML>

```

Error codes

Code	Description
General Return Codes	
0	OK
8001	General error
8002	Operation cancelled by user
8003	Memory error
8004	Invalid parameter
8005	Failed to decode request
8006	Failed to encode request
8007	Failed to convert to/from Unicode
8008	Operation not supported
8009	Token not present
8010	Failed to determine page URL
8011	Server not trusted
8012	Parameter is not Boolean
8013	Incorrect PIN
8022	Plug-in cannot communicate with Personal
PIN Administration Return Codes	
8101	PIN policy error
8102	PIN is blocked
8103	PIN is not blocked
8104	Incorrect PUK
Export Return Codes	
8201	Error writing exported token
Import Return Codes	
8301	Failed to import token
Delete Return Codes	
8401	Failed to delete token(s)
Renew Polldates Return Codes	
8601	Auto-update Manager not present.

Configuration

The branding DLL contains the hosts that are allowed to run this plug-in. If no host configuration exists, all hosts are allowed to run this plug-in.

Chapter 7. LogoutTokens Plug-in

Introduction

The LogoutTokens plug-in allows the web server to log out a user from the token once the session has been completed.

After the user has finished the session and logs out from the current web site, the web server invalidates the session key by clearing it at the server side. The log out from the server prevents someone else from creating a new session without entering the PIN again.

The plug-in must be scripted. This means that, after the plug-in has been activated, a script function must be called in order to log out the user from the token.

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	B2D171C8-6B69-487D-9267-806486775771 (<i>Windows only</i>)
ProgID	Nexus.LogoutCtl (<i>Windows only</i>)
Activation MIME type	application/x-personal-logout

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It is activated using the <OBJECT> tag.

Example 7.1. Example of an ActiveX control activation

```
<OBJECT ID="LogoutTokens"
        CLASSID="CLSID:B2D171C8-6B69-487D-9267806486775771">
</OBJECT>
```

It is also possible for the web server to use a scripting language to silently detect if the plug-in is installed in the client.

Example 7.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.LogoutCtl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

There are no issues with Eolas' patent since the plug-in does not use any parameters.

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the <OBJECT> tag. It should be noted that this is done in a different way than for Internet Explorer. The ClassID is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 7.3. Example of how to activate the Mozilla-based browsers plug-in using the <OBJECT> tag

```
<OBJECT ID="Logout" TYPE="application/x-personal-logout">
</OBJECT>
```

A script language can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 7.4. Example of a script to activate the plug-in:

```
<SCRIPT language="JavaScript">
  if(navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personallogout"]) {
        if (navigator
            .mimeTypes["application/x-personal-logout"].enabledPlugin)
        {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

The plug-in does not take any parameters.

Scripting

The following functions are implemented

LogoutTokens

```
int LogoutTokens();
```

Logs out the user from all logged in tokens.

Always returns 0 (i.e. successful).

Usage

The web server should call this plug-in right before it invalidates the SSL session key. Then it will not be possible to create a new session key, without the user having to enter the PIN again.

The plug-in knows in which process it is running. That is the same browser process that has loaded the CSP (*Windows only*) or PKCS#11 library and logged in to a token during the SSL client authentication.

The plug-in then can send this information to the Personal application, which can send an event telling the appropriate process to log out of its tokens.

However, it cannot tell exactly which token to logout, so all logged in tokens in that process have to log out. This will not be a problem in most cases, since the browser usually can only handle one SSL session, and the plug-ins will not be constantly logged in. Thus, no other token should be logged in, to that process.

Sample Web Pages

This section includes some basic sample pages, showing how to activate the plug-in. For more extensive samples, see the sample pages package.

Example 7.5. Example of how to Detect and Activate the Plug-in in Internet Explorer (*Windows only*)

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.LogoutCtl");
      if(xObj) {
        document.writeln("Object installed.");
      }
      document.writeln("Object not installed.");
    }
  </SCRIPT>

  <OBJECT ID="LogoutTokens"
    CLASSID="CLSID:B2D171C8-6B69-487D-9267806486775771">
  </OBJECT>
  <SCRIPT language="JavaScript">
    LogoutTokens.LogoutTokens();
  </SCRIPT>
</HTML>
```

Example 7.6. Example of how to Detect and Activate the Plug-in in Mozilla-based browsers

```
<HTML>
  <SCRIPT language="JavaScript">
    if(navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
            navigator.mimeTypes["application/x-personallogout"]) {
          if (navigator
              .mimeTypes["application/x-personal-logout"].enabledPlugin) {
            document.writeln("Plugin installed");
          }
        }
      }
    }
  </SCRIPT>

  <OBJECT id="logoutId" type="application/x-personal-logout">
</OBJECT>

  <SCRIPT language="JavaScript">
    var logout = document.getElementById('logoutId');
    logout.LogoutTokens();
  </SCRIPT>
</HTML>
```

Security Issues

The plug-in might open up for denial-of-service attacks, allowing malicious web content to log out the user from the currently logged in tokens. However, this requires the user to view the malicious content in the same process as the one currently logged into using client authenticated SSL. Even if this would happen, the effect is quite harmless.

Chapter 8. Version Plug-in

Introduction

This plug-in is used to retrieve the version of Personal and its installed components.

Plug-in Activation

The following <OBJECT> tags are used to activate the plug-in in a web browser:

ClassID	E5C324CC-4029-43CA-8D57-4A10480B9016	(Windows only)
ProgID	Nexus.VersionCtl	(Windows only)
Activation MIME type	application/x-personal-version	

Internet Explorer

Note

This section applies to the Windows platform only.

In Internet Explorer, the plug-in is implemented as an ActiveX control. It can be activated using the <OBJECT> tag.

Example 8.1. Example of an ActiveX control activation

```
<OBJECT ID="PersonalVersion"
        CLASSID="CLSID:E5C324CC-4029-43CA-8D574A10480B9016">
</OBJECT>
```

It is also possible for the web server to use scripting to silently detect if the plug-in is installed in the client.

Example 8.2. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  try {
    var xObj = new ActiveXObject("Nexus.VersionCtl");
    if(xObj) {
      document.writeln("Object installed.");
    }
  } catch (e) {
    document.writeln("Object not installed.");
  }
</SCRIPT>
```

If a scripting language is used, we recommend that solution 1 be used. Please refer to “Appendix A — Eolas Patent” for more information.

Mozilla-Based Browsers

In Mozilla-based browsers, the plug-in is implemented using the NPAPI. It can be activated using the `<OBJECT>` tag. It should be noted that this is done in a different way than for Internet Explorer. The `ClassID` is not used to identify the plug-in, but rather the activation MIME type as defined above.

Example 8.3. Example of how to activate the Mozilla-based browsers plug-in using the `<OBJECT>` tag

```
<OBJECT ID="Version" TYPE="application/x-personal-version">
</OBJECT>
```

A script language can be used by the web server to decide whether the plug-in is installed in the browser by checking if the activation MIME type is registered.

Example 8.4. Example of a script to activate the plug-in

```
<SCRIPT language="JavaScript">
  if(navigator.plugins) {
    if (navigator.plugins.length > 0) {
      if (navigator.mimeTypes &&
          navigator.mimeTypes["application/x-personalversion"]) {
        if (navigator.mimeTypes["application/x-personal-version"].enabledPlugin)
        {
          document.writeln("Plugin installed");
        }
      }
    }
  }
</SCRIPT>
```

Parameters

Parameter	Explanation
<i>PostUrl</i>	Sets the URL to which the plug-in should post its data. If this parameter is not set, no data will be posted. An absolute URL is required.

Scripting

The following functions are implemented

```
PostVersion          void PostVersion(postUrl);

                     String postUrl;
```

If the plug-in is activated without setting the *PostUrl* parameter in a `<PARAM>` tag, the plug-in will not post any data. The `PostVersion` method can be used to post the version at a later stage.

GetVersion

string **GetVersion()**;

Returns String

Returns the version string without posting it. The format of the string is described below.

Sample Web Pages

This section includes some basic sample pages, showing how to activate the plug-in. For more extensive samples, see the sample pages package.

Example 8.5. Example of Direct Activation using Internet Explorer (*Windows only*):

```
<HTML>
  <OBJECT ID="PersonalVersion"
    CLASSID="CLSID:E5C324CC-4029-43CA-8D574A10480B9016">
    <PARAM NAME='PostURL' VALUE='http://server.com/post'>
  </OBJECT>
</HTML>
```

Example 8.6. Example of Scripting using Internet Explorer (*Windows only*)

```
<HTML>
  <SCRIPT language="JavaScript">
    try {
      var xObj = new ActiveXObject("Nexus.VersionCtl");
      if(xObj) {
        document.writeln("Object installed.");
      }
    } catch (e) {
      document.writeln("Object not installed.");
    }
  </SCRIPT>
  <OBJECT ID="version"
    CLASSID="CLSID:E5C324CC-4029-43CA-8D57-4A10480B9016">
  </OBJECT>
  <SCRIPT language="JavaScript">
    document.writeln(version.GetVersion());
  </SCRIPT>
</HTML>
```

Example 8.7. Examples of Direct Activation using Mozilla-based browsers

```
<HTML>
  <OBJECT ID="PersonalVersion" type="application/x-personal-version">
    <PARAM NAME='PostURL' VALUE='http://server.com/post'>
  </OBJECT>
</HTML>
```

Example 8.8. Example of Scripting using Mozilla-based browsers

```

<HTML>
  <SCRIPT language="JavaScript">
    if(navigator.plugins) {
      if (navigator.plugins.length > 0) {
        if (navigator.mimeTypes &&
            navigator.mimeTypes["application/x-personalversion"]) {
          if (navigator.mimeTypes["application/x-personal-version"].enabledPlugin) {
            document.writeln("Plugin installed");
          }
        }
      }
    }
  </SCRIPT>
  <OBJECT ID="versionId"
          TYPE="application/x-personal-version"
          LENGTH=0
          HEIGHT=0>
  </OBJECT>
  <SCRIPT language="JavaScript">
    var version = document.getElementById('versionId');
    document.writeln(version.GetVersion());
  </SCRIPT>
</HTML>

```

Output format

The version plug-in will enumerate all installed components and either send them as a web form, i.e. Content-Type: application/x-www-form-urlencoded, using HTTP Post or make them available for the GetVersion() method.

The body of the post, or the return value of the GetVersion() method may contain the following items.

1. Version of Personal in the format (mandatory)

```
Personal=<version nr>
```

2. Version of each installed component in the format

```
<file name>_<extension>=<version nr>
```

Multiple modules are separated by "&"

3. The string CSP_INSTALLED={TRUE, FALSE}, specifying if the CSP is installed or not. (*Windows only*)

4. Smart Card readers available for Personal in the format:

```
SmartCard_Reader=<reader name>
```

If multiple readers are installed, several of these entries are returned. The fields are *not* to be treated as case sensitive.

5. Platform

Platform is either win32, win64, macosx or linux.

6. OS version

```
For platform win32      os_version={win95, win98, winme, winnt,
win2000, win2003, winxp, winvista, win7,
unknown}
```

For platform macosx `os_version=<majorversion>.<minorversion>`
I.e. 10.4, 10.5 etc. or unknown.

For platform linux `os_version=8.04`
Specifying the Ubuntu distribution version.

7. Distribution `distribution=ubuntu`

Specifying the os distribution. (Linux Only)

8. Time when Personal at the latest should do an update check

`best_before=<value of best-before-date>`

9. File signing capability of Personal.

`docSign=1`

`docSign=1` is the only allowed value if filesigning is supported.

10. Unique hardware identification string.

`uhi=<base64 encoded string>`

Example 8.9. Example of a version string

```
Personal_exe=4.10.0.33&persinst_exe=4.10.0.33&ctest_ng_exe=1.0.0.1&tokenapi_dll=4.10.0.29&personal_dll=4.10.0.29&np_prsnl_dll=4.10.0.33&lng_svse_dll=4.10.0.33&crdsiem_dll=4.10.0.33&crdsetec_dll=4.10.0.33&crdprism_dll=4.10.0.33&br_svse_dll=1.4.0.9&br_enu_dll=1.4.0.9&branding_dll=1.4.0.9&CSP_INSTALLED=TRUE&Personal=4.10.0.33&platform=win32&os_version=winxp&best_before=1224740253&docSign=1&uhi=qFFzPs1w111e0YuJf08h+yOerhg=&
```

Chapter 9. Personal PKCS#11

Introduction

This chapter contains information about the PKCS#11 module in Personal.

The interface to Personal PKCS#11 is compatible with Cryptoki (PKCS #11 version v2.20) and implements a subset of the API as defined in reference [6].

This module is installed in the Mozilla-based browsers in order to enable SSL authentication.

Mozilla Browsers

The Personal PKCS#11 module is automatically registered as a cryptographic module for all found Mozilla user profiles.

PKCS #11 API

This chapter describes the API functions implemented in the Personal PKCS#11 module. Only the functions that have been implemented have been detailed below.

General Purpose

Function	Description
C_Initialize	Initializes Cryptoki.
C_Finalize	Cleans up miscellaneous Cryptoki functions.
C_GetInfo	Obtains general information about Cryptoki.
C_GetFunctionList	Obtains entry points of Cryptoki library functions.

Slot and Token Management

Function	Description
C_GetSlotList	Obtains a list of slots in the system.
C_GetSlotInfo	Obtains information about a particular slot.
C_GetTokenInfo	Obtains information about a particular token.
C_GetMechanismList	Obtains a list of mechanisms supported by a token.
C_GetMechanismInfo	Obtains information about a particular mechanism.
C_SetPIN	Modifies the PIN of the user that is currently logged in.
C_WaitForSlotEvent	Waits for slot event (token insertion, removal etc.) to occur.
C_UnblockPIN	This function is not member of the PKCS #11 v2.20 specification. It can be used to unblock a PIN.

Session Management

Function	Description
C_OpenSession	Opens a connection or “session” between an application and a particular token.
C_CloseSession	Closes a session.
C_CloseAllSessions	Closes all sessions with a token.
C_GetSessionInfo	Obtains information about the session.
C_Login	Logs into a token. Only CKU_USER is supported for soft tokens.
C_Logout	Logs out from a token.

Object Management

Function	Description
C_CreateObject	Creates a new object.
C_CopyObject	Copies an object, creating a new object for the copy.
C_DestroyObject	Destroys an object.
C_GetAttributeValue	Obtains an attribute value of an object.
C_SetAttributeValue	Modifies the value of one or more attributes of an object.
C_FindObjectsInit	Initializes an object search operation.
C_FindObjects	Continues an object search operation.
C_FindObjectsFinal	Finishes an object search operation.

Encryption and Decryption

Function	Description
C_EncryptInit	Initializes an encryption operation.
C_Encrypt	Encrypts single-part data.
C_EncryptUpdate	Continues a multiple-part encryption operation.
C_EncryptFinal	Finishes a multiple-part encryption operation.
C_DecryptInit	Initializes a decryption operation.
C_Decrypt	Decrypts single-part encrypted data.
C_DecryptUpdate	Continues a multiple-part decryption operation.
C_DecryptFinal	Finishes a multiple-part decryption operation.

Message Digesting

Function	Description
C_DigestInit	Initializes a message — digesting operation.
C_Digest	Digests data in a single part.
C_DigestUpdate	Continues a multiple-part digesting operation.
C_DigestFinal	Finishes a multiple-part digesting operation.

Signing and Verifying

Function	Description
C_SignInit	Initializes a signature operation.
C_Sign	Signs single part data.
C_SignUpdate	Continues a multiple-part signing operation.
C_SignFinal	Finishes a multiple-part signing operation.
C_SignRecoverInit	Initializes a signature operation, where the data can be recovered from the signature.
C_SignRecover	Signs single-part data, where the data can be recovered from the signature.
C_VerifyInit	Initializes a verification operation.
C_Verify	Verifies a signature on single-part data.
C_VerifyRecoverInit	Initializes a verification operation where the data is recovered from the signature.
C_VerifyRecover	Verifies single-part data, where the data can be recovered from the signature.

Key Management

Function	Description
C_GenerateKeyPair	Generates a public/private key pair, creating new key objects.
C_WrapKey	Wraps (i.e. encrypts) a key.
C_UnwrapKey	Unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object.

Random Number Generation

Function	Description
C_SeedRandom	Mixes additional seed material into the token's random number generator.
C_GenerateRandom	Generates random or pseudo- random data.

Interoperability

The PKCS #11 v2.20 specification is the complete documentation of the API implemented by Personal PKCS#11. This section only describes exceptions from PKCS #11 specification.

C_GetSlotList()	Each physical smart card token will, by default, be shown as two virtual tokens to allow multiple PIN codes on a smart card. If only one PIN is available, then the second slot will be empty.
C_Login()	Only CKU_USER is supported for software tokens.
C_FindObjectsInit()	A return value is added: CKR_TEMPLATE_INCONSISTENT
C_DecryptUpdate()	A return value is added: CKR_FUNCTION_NOT_PERMITTED

C_EncryptUpdate() A return value is added:
CKR_FUNCTION_NOT_PERMITTED

C_UnblockPIN() This function is not member of the PKCS #11 v2.20 specification. It can be used to unblock a locked PIN.

```
CK_RV C_UnblockPIN( CK_SESSION_HANDLE hSession,  
                    CK_CHAR_PTR pPin,  
                    CK_ULONG ulPinLen,  
                    CK_CHAR_PTR pPuk,  
                    CK_ULONG ulPukLen) ;
```

hSession is an ordinary session handle.

pPin should point to a buffer containing the PIN. It depends on the token whether this should be the old PIN or a new PIN.

pPuk is a pointer to the unblocking code.

PKCS#11 Configuration

It is possible to configure the PKCS#11 to customer specific needs. See “ Appendix D — CSP and PKCS#11 Configuring”.

Chapter 10. Personal CSP

Introduction

Note

This chapter applies to the Windows platform only.

This chapter contains information about the CSP module in Personal.

The CSP component of Personal implements a Microsoft CSP (Cryptographic Service Provider).

Any application using MSCAPI can automatically access the smart card and smart card reader support of Personal. Some examples of applications using this API are WinLogon, Microsoft Office 2003 (and later), Internet Explorer, and various VPN clients. The Microsoft ActiveX control XEnroll is also using this API.

CSP Information

Personal CSP has the following characteristics:

Provider type	PROV_RSA_FULL
Provider name	Personal CSP

CSP Functions

This chapter describes the functions implemented in Personal CSP.

A detailed description of the various Microsoft CryptoAPI functions can be found at the MSDN Library, see reference [6].

Section “Interoperability ” specifies any deviations in the Personal CSP implementation.

CSP Connection

Function	Description
CPAcquireContext	Acquires a handle to a key container in a particular CSP.
CPGetProvParam	Retrieves attributes from Personal CSP.
CPReleaseContext	Releases the handle acquired by the CryptAcquireContext function.
CPSetProvParam	Specifies attributes of a CSP.

Key Management

Function	Description
CPDestroyKey	Destroys or releases a handle to a key.
CPExportKey	Transfers a key from the CSP to a key BLOB in the application's memory space.
CPGenKey	Generates a random key.
CPGetUserKey	Gets a handle to the key exchange or signature key.

Function	Description
CPGenRandom	Generates random data.
CPGetKeyParam	Retrieves the parameters of a key.
CPImportKey	Transfers a key from a key BLOB to a CSP.
CPSetKeyParam	Specifies the parameters of a key.

Hashing and Digital Signatures

Function	Description
CPCreateHash	Creates an empty hash object.
CPGetHashParam	Retrieves a hash object parameter.
CPHashData	Hashes a block of data and adds it to the specified hash object.
CPSetHashParam	Sets a hash object parameter.
CPSignHash	Signs the specified hash object.

Encryption

Function	Description
CPDecrypt	Decrypts a section of plaintext by using the specified encryption key.

Interoperability

CPAcquireContext()

The flags CRYPT_VERIFYCONTEXT, CRYPT_SILENT, CRYPT_DELETEKEYSET, and CRYPT_NEWKEYSET are supported. The flag CRYPT_MACHINE_KEYSET is ignored.

If this method is called with container parameter = NULL, the default key container will be used. The default key container in the configuration file will be used if specified, otherwise, CSP will try to create a context according to the following principles:

1. Use the first card reader having a smart card inserted.
2. Use the first card reader without a card.
3. A context will be created like if CPAcquireContext is called with the flag CRYPT_VERIFYCONTEXT.

Personal supports the following types of container names:

- Card reader containers using the Microsofts format: \\.\<reader_name>\.

Example: \\.\Gemplus USB Smart Card Reader 0\.

- A container representing a certificate. The container name is then the SHA-1 hash of the certificate.

Example:

90C75B312BEE4F8117EA90EDC7F8F395314ECFEA

- Other supported container names, described in “Container Name”, are
 - GUID (Globally Unique Identifier)
 - \\.\<reader>\<id> and \\.\<reader>\0x<id>

CPGetProvParam()

The query PP_KEYSET_SEC_DESCR is *not* supported.

If a CSP function returns NTE_FAIL, it is possible to call CryptGetProvParam with dwParam=PP_NX_LAST_ERROR in order to get a more detailed explanation of the error situation.

PP_NX_LAST_ERROR	0x80000001
------------------	------------

The following error codes are defined:

ERROR_NX_UNDEFINED	0x80007000
ERROR_NX_PIN_INCORRECT	0x80007001
ERROR_NX_PIN_BLOCKED	0x80007002

For each call to a CSP API function, the error code is reset to ERROR_NX_UNDEFINED.

Note

The error codes are set per thread. If several threads are running, it is only the one that got NTE_FAIL in return from CSP that can call CryptGetProvParam to get PP_NX_LAST_ERROR.

Parameter PP_NX_CONTEXT_FLAGS can be used to get more information about the token.

NX_FLAG_HW_TOKEN_PRESENT	0x00000001	<text>Smart card token</text>
NX_FLAG_HAS_PROTECTED_PATH	0x00000002	<text>PIN-Pad reader</text>
NX_FLAG_TOKEN_REMOVABLE	0x00000004	<text>Token on removable media</text>
NX_FLAG_TOKEN_PRESENT	0x00000008	<text>Token is available</text>

CPSetProvParam()

Parameter PP_KEYSET_SEC_DESCR is not supported. Additional supported parameters are: PP_KEYEXCHANGE_PIN (value 32) and PP_SIGNATURE_PIN (value 33).

pbData should point to a string containing the PIN.

CPExportKey()

OPAQUEKEYBLOB and PRIVATEKEYBLOB are *not* supported.

CPGenKey()

If algorithm identifier AT_KEYEXCHANGE or AT_SIGNATURE is used, the flag

	CRYPT_USER_PROTECTED has to be set. The flag CRYPT_EXPORTABLE must not be set. Supported key lengths are 512, 768, 1024, and 2048 bits.
CPGetKeyParam()	Additional parameter value supported is KP_CERTIFICATE (value 26), which, if successful, will return a certificate associated with the key.
CPImportKey()	Supported key BLOB types are PUBLICKEYBLOB, SIMPLEBLOB and PRIVATEKEYBLOB.
CPSetKeyParam()	Additional parameter value supported is KP_CERTIFICATE (value 26). When using this parameter, a certificate is stored on the token having the key associated with this certificate.
CPCreateHash()	Supported mechanisms are CALG_SSL3_SHAMD5, CALG_MD5 and CALG_SHA1.
CPHashData()	CRYPT_USERDATA is <i>not</i> supported.
CPSetHashParam()	Supported parameter is HP_HASHVAL.
CPSignHash()	The flag CRYPT_NOHASHOID is supported for all tokens that support RSA with PKCS#1 padding. All soft tokens support this, and the requirement for cards is that no hash OID is added to the data that is signed.

Additional Comments

The Personal application (personal.exe) moves the certificates from the smart card or the software token to MS Cert Store, where they can be accessed by e.g. Internet Explorer.

Using XEnroll with Personal CSP

XEnroll, which is an ActiveX component from Microsoft, can be used with Personal CSP for certificate enrollment. ICEnroll is a set of interfaces used with XEnroll.

The XEnroll interface has a property, UseExistingKeySet, which should be set to TRUE if secondary certificates are to be issued. If the property is set to FALSE, new keys should always be generated.

In XEnroll, the container name is set by the property ContainerName.

Container Name

Container name can be specified in various ways:

1. Blank container name
2. Card reader name
3. Card reader name and ID
4. One, by Personal unknown, container name

Blank container name

If a blank container name is set, the default container will be selected. The default container is the first card reader (with an inserted smart card) detected by the CSP. If no card is inserted, the first card

reader detected will be selected. XEnroll will call the CSP with an autogenerated container name in GUID format, e.g., 857b6eeb-5f4c-4edb-b87e-07b883c205d3.

If a key is to be generated when the request is created, it is only possible to generate one key of each sort, i.e. one AT_KEYEXCHANGE and one AT_SIGNATURE. The reason is that the key ID is based on the container name and has to be unique for a key pair on the token.

Card reader name

A card reader name should have the format

```
\\.\<reader name>\
```

Example:

```
\\.\Gemplus USB Smart Card Reader 0\
```

If a key is to be generated when the request is created, it is only possible to generate one key of each sort, i.e. one AT_KEYEXCHANGE and one AT_SIGNATURE. The reason is that the key ID is based on the container name and has to be unique for a key pair on the token.

Card reader name and ID

Use one of the following formats:

```
\\.\<reader name>\<id>
```

or

```
\\.\<reader name>\0x<id>
```

Example 1:

```
\\.\Gemplus USB SmartCard Reader 0\1
```

In this case ID will be 0x31.

Example 2:

```
\\.\Gemplus USB SmartCard Reader 0\0x55
```

In this case ID will be 0x55.

When this format is used, it is possible to control for which key a certificate should be issued. If option key generation is selected, the key ID will be the ID of the container name. If an existing key is used, its ID should be known.

If this format is used, and a card reader, unknown to Personal, is specified, the default container will be used and a key pair with the given ID will be searched for or created.

Unknown container name

Create new key set
(UseExistingKeySet=0)

If the CSP does not find a key/token with the specified container name, the default container will be selected. Unknown container name means that the format does not comply with the possible ones (see “CPAcquireContext”)

Use existing key set
(UseExistingKeySet=1)

If the key/token is not found, the CSP will return an error code.

Example

This example contains an HTML page using XEnroll. An authentication key is generated and a certificate is requested for that key.


```

<HTML>
<OBJECT NAME="XEnrollObject"
        CLASSID="CLSID:127698e4-e730-4e5c-a2b1-21490a70c8a1">
</OBJECT>

<SCRIPT Language="Javascript">
function doRequest() {
    var enrollment = document.XEnrollObject;
    enrollment.ProviderType= '1'
    enrollment.ProviderName= 'Personal CSP'
    enrollment.UseExistingKeySet= '0'
    enrollment.GenKeyFlags= '2'
    enrollment.ContainerName= '\\\\.\Gemplus GemPC430 0\\0x22'
    enrollment.HashAlgorithm= 'SHA1'
    enrollment.KeySpec= '1'
    document.Order.pkcs10.value = enrollment.CreatePKCS10("CN=cm", "")
    document.Order.submit();
}
</SCRIPT>
<FORM NAME="Order"
        ACTION="/servlet/XEnroll"
        ENCTYPE=x-www-form-encoded
        METHOD="POST">
<INPUT TYPE="hidden"
        NAME="pkcs10"
        VALUE="">
<INPUT TYPE="hidden"
        NAME="Process"
        VALUE="order">
<INPUT TYPE="hidden"
        NAME="Response"
        VALUE="order/exp">
<TABLE BORDER=1 CELLPADDING=6 BGCOLOR="#CCCCCC" >
<TR>
<TD>
<B>First Name</B>
</TD>
<TD>
<INPUT NAME="firstname"
        TYPE="text"
        SIZE="30">
</TD>
</TR>
<TR>
<TD>
<B>Last Name</B>
</TD>
<TD>
<INPUT NAME="lastname"
        TYPE="text"
        SIZE="30">
</TD>
</TR>
</TABLE>
<BR>
<INPUT TYPE="button"
        NAME="SubmitApp"
        VALUE="Submit Application"
        onClick="doRequest()">
</FORM>
</HTML>

```

CSP Configuration

It is possible to configure the CSP to customer specific needs. See “Appendix D — CSP and PKCS#11 Configuring”.

Chapter 11. Installation on Windows

Program Requirements

The overall goals for the packaging and distribution of Personal are as follows:

- Installation should be a "one-click" procedure.
- Keep it small and adapted to large web-based installations.
- Support for pushing out installations/upgrades to users within an organization.

In order to reach these goals, a dedicated installation program performs the various aspects of the installation. Furthermore, the installation is packaged into browser specific packages for easy installation from web-based environments. The installation also manages PKCS#11, CSP, ActiveX, and plug-in installation.

The Installation Program

The installation program is a standard Windows executable. All the files to be installed are packaged into the executable `persinst.exe`.

Packaging

The installation program `persinst.exe` is packaged for four different distributions.

- A signed CAB file for installation in Internet Explorer.
- An XPI file for installation in Mozilla-based web browsers.
- An EXE file created using the IExpress packager for traditional distribution. The EXE file is signed and compressed.
- A copy of `persinst.exe`, which can be used when building customer specific packets based on Personal.

Installation Conditions

Installation and uninstallation of Personal is performed by the program `persinst.exe`. The behavior of the program is controlled by parameters in the optional file `persinst.cfg`, which must be stored in the same directory. A default installation will take place if no `persinst.cfg` is available. In addition, if a file named `personal.cfg` exists in the same directory, it will also replace the shipped `personal.cfg` located in PersonalInstall (see below) during the installation.

`persinst.cfg` enables an organisation to customize the installation process to suit special needs.

`personal.cfg` enables an organisation to provide settings that will make Personal behave in a predetermined way.

Note

The delivered file `personal.cfg` is a sample configuration file containing all parameters not described elsewhere. See the well commented file, for detailed information.

By default, “master copies” of the program and configuration files are stored in `PersonalInstall\bin`, a location defined by the registry key `HKEY_LOCAL_MACHINE\Software\Personal\<version>\PersonalInstall` . (Normally `C:\Program Files\Personal`).

When a user starts Personal, a private configuration file is stored in `PersonalHome\config`, a location defined by the registry key `HKEY_CURRENT_USER\Software\Personal\<version>\PersonalHome` . (Normally `C:\Documents and Settings\<user_name>\Application Data\Personal`.)

Each time a user starts Personal, a check is made against `InstallUid` in the registry to see if the master copy has been updated. If so, the user’s private copy of the configuration file `personal.cfg` will be merged with any new settings from the master copy.

An example of `personal.cfg` is shipped with the product. Use this file to find out what parameters can be set.

Installation Configuration

All parameters in `persinst.cfg` are well commented and therefore not described in detail in this document. The following options exist:

- To display a message or open a URL after a successful installation.
- To use the Event log for logging.
- To perform a silent or non-silent installation.
- To specify the questions and answers to be asked by the installation program.
- To display reference number in error messages.
- To specify if Personal should be added to Add/Remove programs in the Control Panel.
- To add additional files to the installation package.

Installation Options

An installation can be silent, i.e., no user action is required. The installation program can perform the following actions:

- Install - install current version of Personal.
- Reinstall - install current version of Personal without uninstalling any existing version.
- Upgrade - uninstall older version of Personal and install current version.
- Modular upgrade - is only able to upgrade the same version. It can be used to install selective files. As an example, this option, which requires a special build of `persinst.exe`, could be useful when distributing components related to a specific language within an organization, configuration updates, or extended smart card support.
- Uninstall - uninstall Personal either using the Add/Remove program function in the Control Panel or the command `persinst.exe -u` in the location defined by the registry key `HKEY_LOCAL_MACHINE\Software\Personal\<version>\PersonalInstall\bin`

Messages from the Installation Program

Personal installs shortcuts in the Programs menu. The following shortcuts are installed:

- Personal
- Personal Guide

Personal also installs a shortcut to the executable under the startup menu so that Personal will start when logging into the machine.

Shortcuts

Personal installs shortcuts in the Programs menu. The following shortcuts are installed:

- Personal
- Personal Guide

Personal also installs a shortcut to the executable under the startup menu so that Personal will start when logging into the machine.

Installation Directory Tree

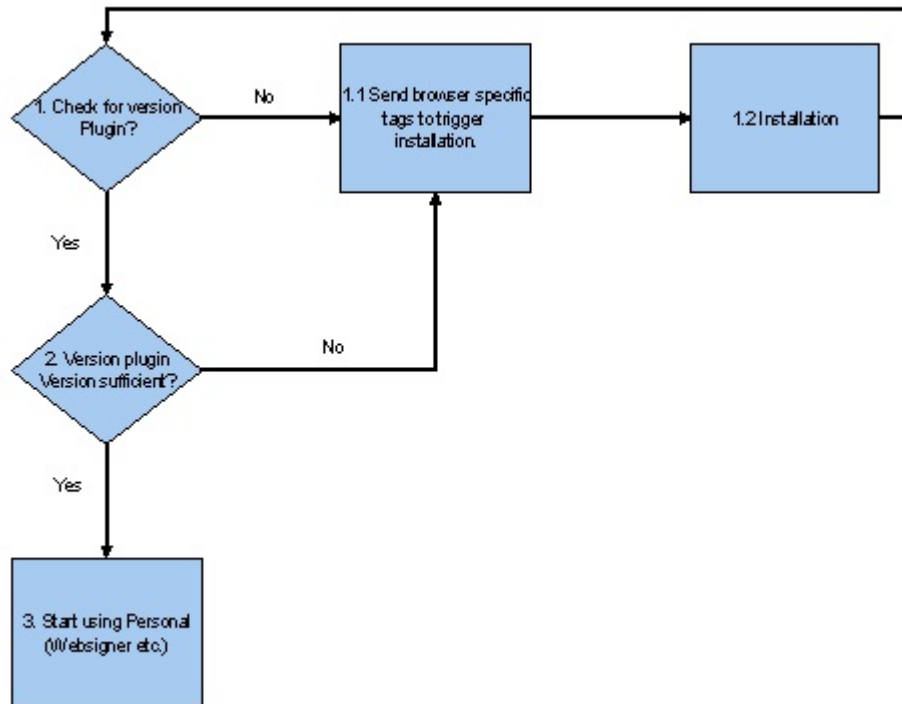
At the time of the installation, the bin, config, and doc directories are created under the `PersonalInstall` key.

The config and store directories are created under the `PersonalHome` key, when Personal is started the first time. In addition, a directory cache is created when Personal is using a smart card token.

<code>bin</code>	The bin directory contains all executables and shared libraries used by Personal. It also contains the signature files for the signed files.
<code>config</code>	The config directory contains the client configuration file <code>personal.cfg</code> .
<code>doc</code>	The doc directory contains the end user help files for all supported languages.
<code>store</code>	The store directory contains the internal software token store.

Web Installation Flowchart

When distributing and installing Personal from a web environment, the following flowchart should be used in order to check if there is a Personal installed and which version resides on the machine.

Figure 11.1. Web Installation Flowchart

1. Check if the version plug-in is present. This is a browser specific operation which is illustrated in the sample JavaScript below. Note that the `<OBJECT>` tag is used without the code base so that the plug-in is not downloaded if it is not found.

Example 11.1.

```

function testForBrowser() {
    var pattern = /netscape/i;
    if (pattern.test(navigator.appName)) {
        browserIsNetscape = true;
    }
    pattern = /microsoft/i;
    if(pattern.test(navigator.appName)) {
        browserIsIE = true;
    }
}
function testForVersionPlugin() {
    if (browserIsIE) {
        try {
            versionPluginObject = new ActiveXObject("Nexus.VersionCtl");
            versionPluginInstalled = (null != versionPluginObject);
        } catch(e) {
            versionPluginObject = null;
        }
    }
    if (browserIsNetscape) {
        if (navigator.mimeTypes["application/x-personal-version"] != null &&
            navigator.mimeTypes["application/x-personal-version"].enabledPlugin != null ) {
            versionPluginObject = null;
            versionPluginInstalled = navigator.mimeTypes["application/x-personal-version"].enabledPlugin;
        }
    }
}
testForBrowser();
if ( browserIsNetscape ) {
    document.write('<OBJECT id="versionId" type="application/x-personal-version"></OBJECT>');
} else if ( browserIsIE ) {
    document.write('<OBJECT id="versionId" CLSID="E5C324CC-4029-43CA-8D57-4A10480B9016"></OBJECT>');
}
testForVersionPlugin()

```

1.1 Sends the tags that trigger the download of the Personal installation. Again this is browser specific, and is illustrated with some sample JavaScripts below. In this step, we use browser specific instantiation of the plug-in object which downloads and installs the CAB or XPI files respectively.

Example 11.2.

```

function installPlugin() {
    var result = false;
    if (browserIsNetscape) {
        xpi={'Personal Signature and Authentication Client':'persinst.xpi'};
        InstallTrigger.install(xpi);
        Install.refreshPlugins();
        result = true;
    }
    if (browserIsIE) {
        document.write('<object classid="CLSID:659D6946-87C7-49a8-BC8A-7579CC223C2A" CODEBASE=persinst.c
        document.write('</object>\n');
        result = true;
    }
    return result;
}

```

Note

The classid above is not that of an existing product, and therefore, it will force an installation of the CAB file.

1.2 Personal will be installed locally. This is handled entirely by the installation program `persinst.exe`. The user can be sent back to step 1.

2. Retrieves the version from the version plug-in. (Refer to “Version Plug-in” on page 81 for more information.) If the Personal version is older than the version on the web page, go to step 1.1 to update to the new version.

3. Personal is installed and the version is sufficient which means that we can proceed and use the Personal plug-ins to enroll and make digital signatures etc.

Limitations

Due to limitations in some browsers, (see `release.txt` for more information), it is necessary to take precautions when designing the web pages. First it is necessary to check the UserAgent string to see if the current browser has such a limitation. This is done by calling a special page. The following example of a JavaScript shows a simple test:

Example 11.3.

```
<script language="Javascript">
if ( ! versionPluginInstalled() )
{
  if (!installPlugin())
  {
    failXpiInstallPatternFF=/Firefox./i;
    failXpiInstallPatternNS=/Netscape.7.2/i;
    if (failXpiInstallPatternFF.test(navigator.userAgent) || failXpiInstallPatternNS.test navigator.
    {
      window.location="index_ns.html";
    }
    else
    {
      window.location='install_cancel.html';
    }
  }
}
</script>
```

The page to be called could look like the following:

Example 11.4.

```
<html>
<head>
<meta http-equiv=Content-Type content="text/html; charset=iso-8859-1">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache"></META>
<title>Personal NG test site</title>
</head>
<a href="persinst.xpi">Installation for Netscape 7.2 and Firefox 1.* users</a>
</body>
</html>
```

CSP & PKCS#11

The installation always copies and registers the file `personal.dll` which contains the PKCS#11 and CSP APIs into the `PersonalHome/bin` directory. If the user has a Mozilla-based web browser installed, the DLL is installed in the browser as a PKCS#11 module.

Plug-ins and ActiveX Controls

The installation always copies the file `np_prsnl.dll` to the `PersonalInstall\bin` directory. The `np_prsnl.dll` file contains all plug-ins and ActiveX objects.

Both plug-ins and ActiveX objects are registered directly to the file under the `PersonalInstall\bin` directory.

Upgrade and Migration

It is possible to upgrade from older versions of Personal to the current version.

Note

The installation program will not uninstall iD2 or SmartTrust versions of Personal but it is possible to migrate soft tokens from those versions to Personal.

Uninstall

Personal is uninstalled through the Add/Remove programs buttons or optionally by executing the `UninstallString` defined by the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\Personal`. If files are locked during uninstallation, the installation program will install itself into the location specified by `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce` and remove the locked files at next reboot.

Controlling the Behavior of Winlogon

In Windows XP, the Windows system process `winlogon.exe` is, by default, configured to read certificates stored on an inserted smart card and put them in the Microsoft Certificate Store.

Personal does the same kind of storing. There is a race condition between `winlogon.exe` and the Personal application, both storing the same certificates in the Certificate Store. If `winlogon.exe` writes the certificates after Personal, some certificate attributes (created by Personal) will be overwritten and Personal will not be able to act properly in some application-specific situations.

However, by changing some Microsoft specific registry settings, it is possible to control the behavior of `winlogon.exe`.

Therefore, the Personal installation program configures `winlogon.exe` not to write certificates to the Certificate Store. It also stores a backup of the original configuration in the Personal specific registry, so it can restore the original `winlogon.exe` configuration if Personal is uninstalled.

The Personal installation program uses this possibility when installing and uninstalling Personal in order to get rid of the described problem.

In Windows Vista, Windows 7 and Windows Server 2008, the Windows function which transfer certificates from a smart card to Microsoft Certificate Store is implemented through domain policies instead. In these environments, the Personal installation program does not take any action to prohibit the Windows behavior and thus the resulting contents of Microsoft Certificate Store it is indeterminate with respect to the certificates on the smart card at hand.

Event Log and Return Codes

Event ID/RC	Text
2000	%1 %2 successfully installed.

Event ID/RC	Text
2001	Operating system version, not supported by installation program.
2002	Installation program requires administrator privileges.
2003	Uninstallation program requires administrator privileges.
2004	A newer version of %1 exists.
2005	Upgrade of %1 declined.
2006	Reinstall of %1 declined.
2007	Upgrade started.
2008	Upgrade failed.
2009	Uninstallation started.
2010	Current installation of %1 cannot be upgraded to current version (%2). Uninstall it and run this installation program again.
2011	Uninstallation started, no installation to uninstall.
2012	Installation started with unknown arguments.
2013	Installation of %1 is already running.
2014	Failure communicating with operating system.
2015	Out of memory.
2016	Failed to create installation directory.
2017	Failed using temporary directory for installation.
2018	Modular upgrade installation does not support switch given to modular upgrade program.
2019	Modular upgrade successfully performed.
2020	Modular upgrade requires administrator privileges.
2021	%1 not installed, modular upgrade cannot be performed.
2022	Modular upgrade of %1 declined.
2023	Modular upgrade of %1 denied. Installed version of %1 was not %2.
2024	Uninstallation of %1 successful.
2025	Uninstallation of %1 successful. Reboot needed.
2026	Installation could not proceed due to problems regarding an existing installation.
2027	Modular upgrade could not proceed due to problems regarding an existing installation.
2028	Operating system version no longer supported by installation program.
2029	Installation could not proceed since a Personal version with incompatible classification is already installed on the system.
4000	Installation of %2 failed. Contact support.

Note

Note: %1 is replaced by product name and %2 is replaced by program version.

Chapter 12. Installation on Macintosh

Introduction

Personal is distributed via a so called disk image file (.dmg), which is opened automatically after downloading into e.g. Safari or Mozilla.

Install

To install Personal, drag and drop Personal.app to the hard drive, e.g. to the desktop or to a location under /Applications.

The installation will be completed when Personal is started the first time. A few messages will be displayed and the user will be informed when the disk image file is opened.

In addition, the following actions are taken:

- PersonalPlugin.bundle (containing the browser plugin) is copied to the users plugin folder ~/Library/Internet Plug-Ins
- The PKCS#11 module, tokenapi.framework, is installed in the browser.
- A folder to store Internal Store is created under ~/Library/Application Support/se.nexus.Personal
- Based on a sample file in Personal.app, an active configuration file is created in ~/Library/Preferences/se.nexus.Personal.cfg
- Temporary files are stored under ~/Library/Caches/se.nexus.Personal

Uninstall

When the user selects Uninstall from the Application menu, Personal will be uninstalled and the following files will be deleted:

- ~/Library/Preferences/se.nexus.Personal.cfg
- ~/Library/Internet Plug-Ins/PersonaPlugin.bundle
- ~/Library/Caches/se.nexus.Personal

Chapter 13. Installation on Linux

Introduction

The installation program consist of a compressed tar file named `personal<VERSION_NUMBER>.tar.gz`.

Install

To install Personal, you need to run the Linux command `tar` to unpack the installation program and then run the installation script `install.sh` as root with the parameter `i`:

```
sudo personal-<VERSION_NR>/install<VERSION_NR>.sh
```

The installation script will perform the following steps:

- All application files, dynamic libraries, language packages, master configuration file, icons and etc are placed under `/usr/local/lib/personal`.
- Soft links to the dynamic libraries are created in `/usr/local/lib`.
- Soft link to the Firefox plug-in library is placed under the designated plug-ins library location for the respective versions of Firefox supported.
- Soft links to the application startup script, `personal.sh` and `persadm.sh`, are placed under `/usr/local/bin/personal`.
- The Nexus Personal Desktop configuration file `personal.desktop` is placed under `/usr/share/applications`. This makes the Personal main application accessible from the Applications menu on the Desktop.
- The `/usr/local/lib` directory is added to the `/etc/ld.so.conf`.

If any of the above fails, the installation will end with an error message.

If Personal is configured to register the PKCS#11 module in Firefox browser it will be done each time Personal starts. To get Personal working with proxy settings, read the comments in the script `/usr/local/lib/personal/personal.sh` how to manually change the proxy environment variables. This is relevant for the parameter `TokenRemovedURL` in the authentication Plug-in if proxy settings are needed to be set.

Uninstall

To uninstall Personal, run the installation script (under `/usr/local/lib/personal`)

```
install.sh
```

This will remove all files installed by the installation program.

Note that the user data located under the directory `.personal` in each user's home directory is not removed.

If the PKCS#11 module is registered in Firefox browser, it has to be manually removed.

Chapter 14. Installation on Citrix

Introduction

Nexus Personal is validated on a Citrix environment based on Citrix Xenapp 6 on Windows 2008 R2, and ICA Client 10 on Windows 7. Citrix supports the use of PC/SC-based cryptographic smart cards, which is the smartcard communication protocol used by Personal.

Install

On such environments, Personal is installed on the server, following regular “ Installation on Windows” instructions. There is off-course no need to install it on the client workstation. The pre-requisites for this will be:

- Activation of smartcard relay in MSTSC (options, local resources, other) [CC note, to be verified]
- Having a running smartcard reader on the workstation

Note

The smartcard reader must be attached before launching the ICA session. When the reader is attached after the ICA session is launched, users must disconnect and relaunch the ICA session to use the smart card inside the session (Refer to <http://support.citrix.com/article/CTX132230> for details)

Nexus do not provide dedicated support for Citrix. More information can be found on Citrix support web site: <http://support.citrix.com/proddocs/topic/xenapp6-w2k8-admin/ps-securing-use-smt-crdw-cps.html>

Chapter 15. Administration

Personal GUI

The User Interface for Personal is used to view and administrate token properties and certificates. The full User Interface is described in this chapter. In summary, the User Interface can be used for the following tasks:

1. Viewing certificates.
2. Viewing tokens.
3. Changing PIN codes.
4. Importing external software tokens.
5. Exporting software tokens to external files.
6. Setting the paths to external PKCS #12 files stored on various media.
7. Changing the settings for web browser integration.

Administration GUI

Personal provides a GUI that is used for administration of the tokens, web browsers, languages and other settings. The administration functions are available in the main application window which can be launched from either Start -> Programs -> Personal <version number> -> Personal, by double-clicking on the Personal tray icon, or by selecting Open command after right-clicking the tray icon.

Figure 15.1. Administration GUI



The main application window can either be displayed in minimized mode where only the task buttons are shown, or in advanced mode where even the tokens are shown. The re-sizing is made either with the button in the lower right corner, or using the command Show tokens in the View menu.

Further, the available tokens can be displayed as small or large sized icons. Small icons are displayed when the command Detailed list is selected from the View menu, and large icons when the command Large icons is selected.

Import and Export

When importing and exporting soft tokens, wizards are launched to guide the user through the process. Soft tokens in PKCS #12 files can be imported to or exported from the Internal Store.

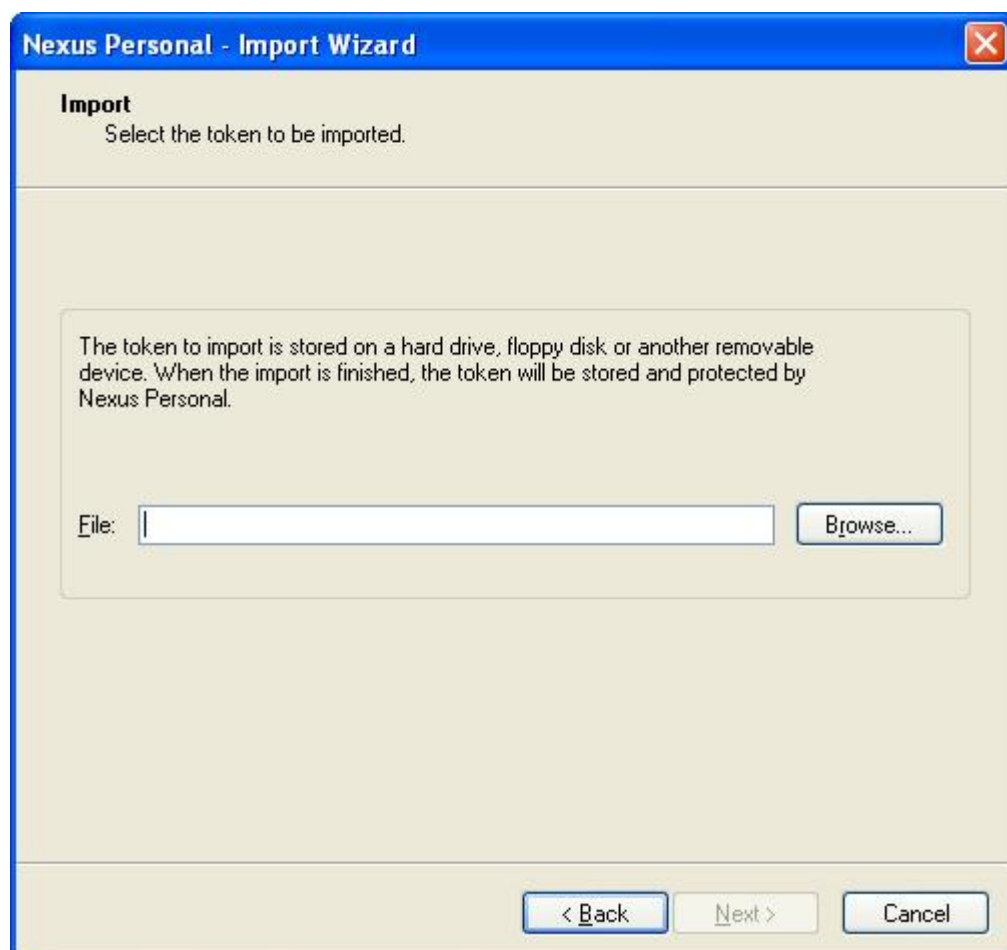
Internal Store

The purpose of having an Internal Store is to prevent attacks where the PKCS #12 file is copied by a virus, trojan horse or downloaded from a shared disk. As the PKCS #12 may be protected by a simple password, it could then be broken by a dictionary attack. The Internal Store adds a secondary encryption layer, based on the user's logon credential (please refer to "Appendix C - Internal Stores" for further details), thus making dictionary attacks more difficult. In order for Personal to be interoperable with other products, it allows import and export of standard PKCS #12 files to and from the Internal Store.

Import Soft Tokens

By using the Import wizard, the PKCS #12 file can be imported into the Internal Store, where the private key is protected by Microsoft Windows Data Protection. The CryptoAPI function CryptProtectData is used to protect the private key. CryptProtectData is used in user mode, meaning that the protection is bound to the user's profile instead of the hardware (please refer to "Appendix C - Internal Stores" for further details). The Import wizard is either started with the Import button in the main application window or from the Import command in the File menu.

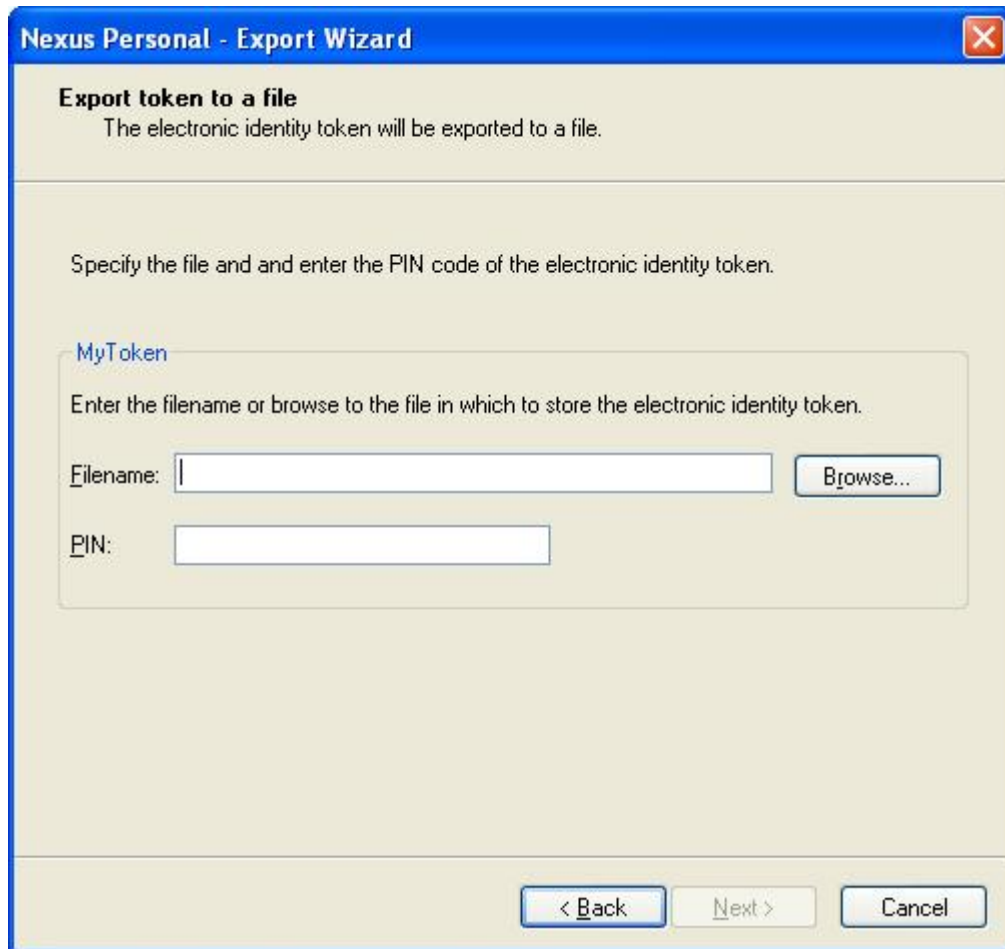
Figure 15.2. Import Soft Tokens



Export Soft Tokens

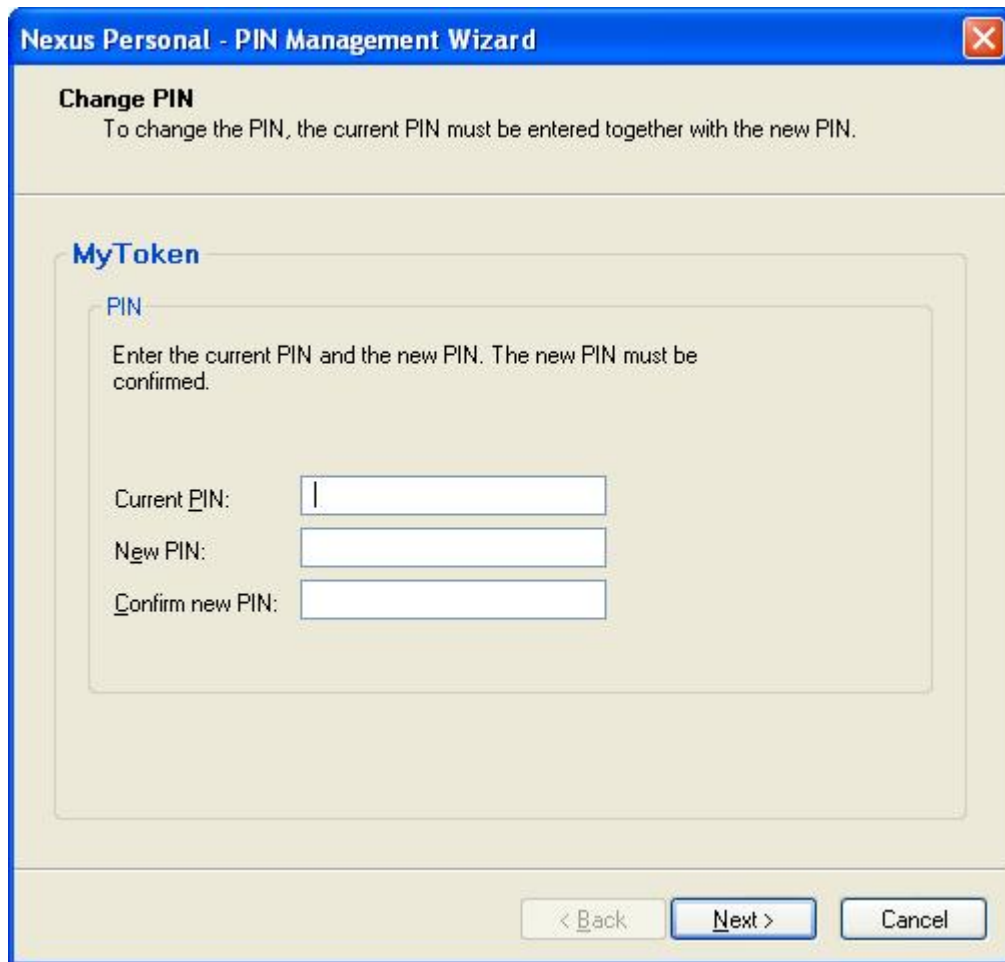
When the Export wizard is used, a protected soft token can be exported into a standard PKCS #12 file. The Export wizard is either started with the Export button in the main application window or from the Export command in the File menu.

Figure 15.3. Export Soft Tokens



Managing PIN Codes

In order to manage the PIN codes of a token, a wizard is launched to allow for either changing or unblocking the PIN code. The unblocking feature is by definition only available for smart cards.

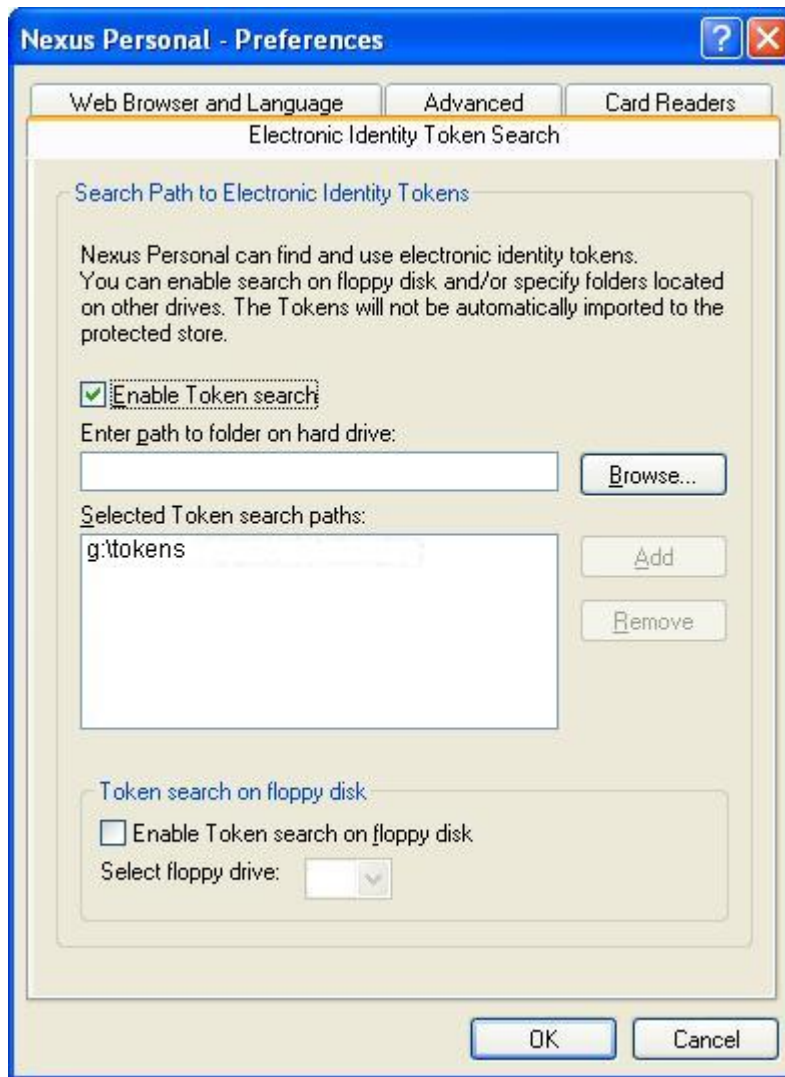
Figure 15.4. Managing PIN Codes

The screenshot shows a dialog box titled "Nexus Personal - PIN Management Wizard". The main heading is "Change PIN" with a sub-instruction: "To change the PIN, the current PIN must be entered together with the new PIN." Below this is a section titled "MyToken" containing a "PIN" sub-section. The instruction reads: "Enter the current PIN and the new PIN. The new PIN must be confirmed." There are three input fields: "Current PIN:" (with a cursor in the first position), "New PIN:", and "Confirm new PIN:". At the bottom right, there are three buttons: "< Back", "Next >", and "Cancel".

The PIN Management Wizard is started with the PIN button in the main application window, from the PIN management command in the File menu, or by right-clicking on an imported soft token and selecting the PIN management command.

Searching for Soft Token

Personal provides functions for mounting drives or directories with stored PKCS #12 files. This feature is available by selecting Preferences in the View menu. In the Preferences window, select the Electronic Identity Token Search tab.

Figure 15.5. Searching for Soft Token

By marking the check box Enable Token search, the mounted drives are scanned for stored PKCS #12 files when the OK button is clicked. By using the Browse... button, drives with PKCS #12 files can be selected. The drive is added to the list of used disks by clicking the Add button, and removed by clicking the Remove button.

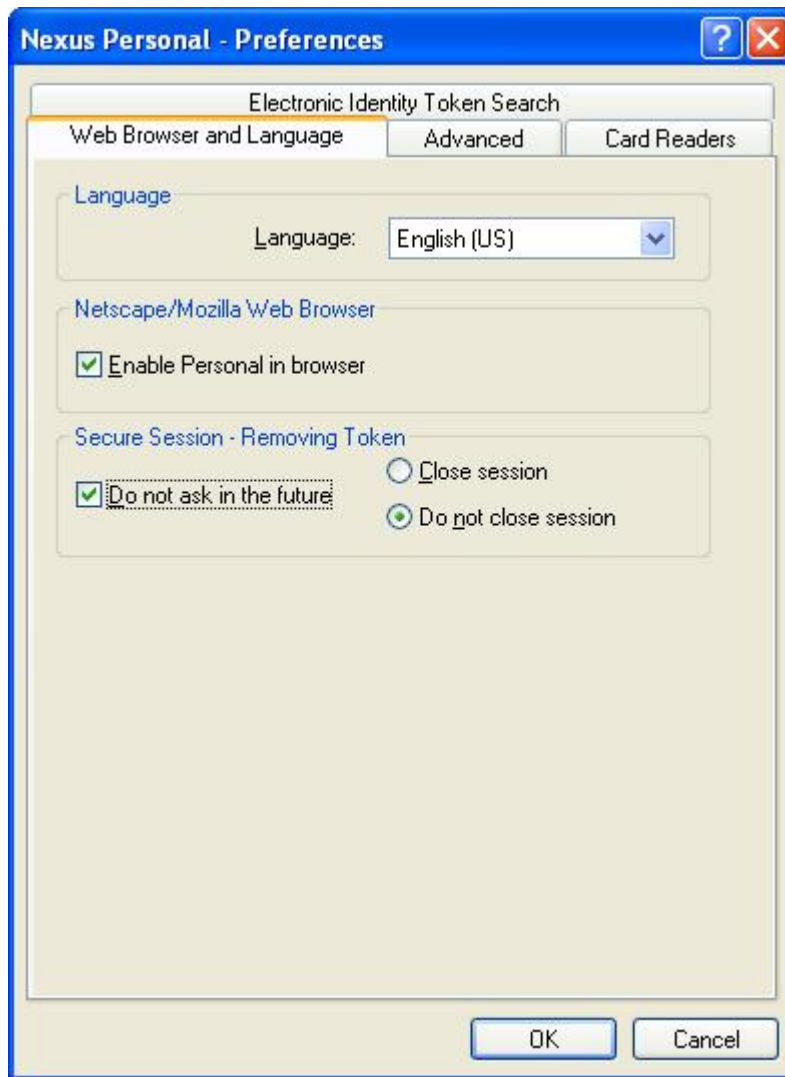
USB drive, CD-ROM, and hard drive paths can be added to the list but not floppy disks. To use a floppy disk, select the option “Enable Token search on floppy disk”. Subdirectories on the floppy disk are not searched.

If the states of the PKCS #12 files on the floppy disk have been changed, Personal is not updated automatically. Select the Refresh command in the View menu to update the tokens in Personal.

This feature facilitates mobility, as PKCS #12 file can be stored on a floppy disk, USB-drive, or CD-ROM, which can be used when travelling. Using a hard drive path allows for interoperability with other PKI clients that may need access to the same PKCS #12 file.

Web Browser and Language Settings

Personal provides functions for configuring the security settings of Mozillabased browsers. This feature is available by selecting Preferences in the View menu. In the Preferences window, select the Web Browser and Language tab.

Figure 15.6. Web Browser and Language Settings

Language

In the Language entry, a preferred language can be chosen. As default, the operating system language is used. If the operating system language is not supported by Personal, English will be chosen as default language in Personal.

Using Personal in the Browser

“Enable Personal in browser” is an option that should be selected if the user has a browser from Mozilla or Firefox. This option will configure these browsers to use the Nexus PKCS#11 cryptographic module.

Normally, during SSL negotiations, these browsers will automatically select an available certificate. Such an action is not wanted. The browsers should always ask the user which certificate to use. This preferred action will be taken if the option "Enable Personal in browser" is selected.

Controlling Secure Sessions in the Browser

NetDetacher is a module in Personal that handles SSL sessions. The behavior of NetDetacher is controlled by settings in this dialog box.

When a token is used in an encrypted and authenticated session (SSL session), the encryption key will be stored in the browser. Even when the token is removed, the session remains valid. This can result in a security flaw unless the session is terminated when the token is removed. As it is not possible to kill only the ongoing session(s), the browser itself has to be terminated.

NetDetacher can be configured to ask the user each time a used token is removed. This feature will be obtained when the option “Do not ask in the future” is not selected. If you do not want to be asked, select this option and, in addition, decide which default action the browser should take, i.e. whether to leave the session or to close it.

Advanced

Personal provides functions for getting information about installed operating system, browsers, and log settings. These features are available by selecting the Preferences in the View menu. In the Preferences window, select the Advanced tab.

Figure 15.7. Advanced

Logging

In the Logging entry, trace files can be enabled. By clicking the Browse button, a directory can be selected for the three trace files that will be created:

- In the file `Nx_prs.log`, all operations carried out in the Token API are logged.
- In the file `Nx_csp.log`, all operations carried out in Personal CSP are logged.
- In the file `Nx_p11.log`, all operations carried out in Personal PKCS#11 are logged.

Operating System

In the Operating System entry, information on the installed operating system and patches is displayed.

Installed Web Browsers

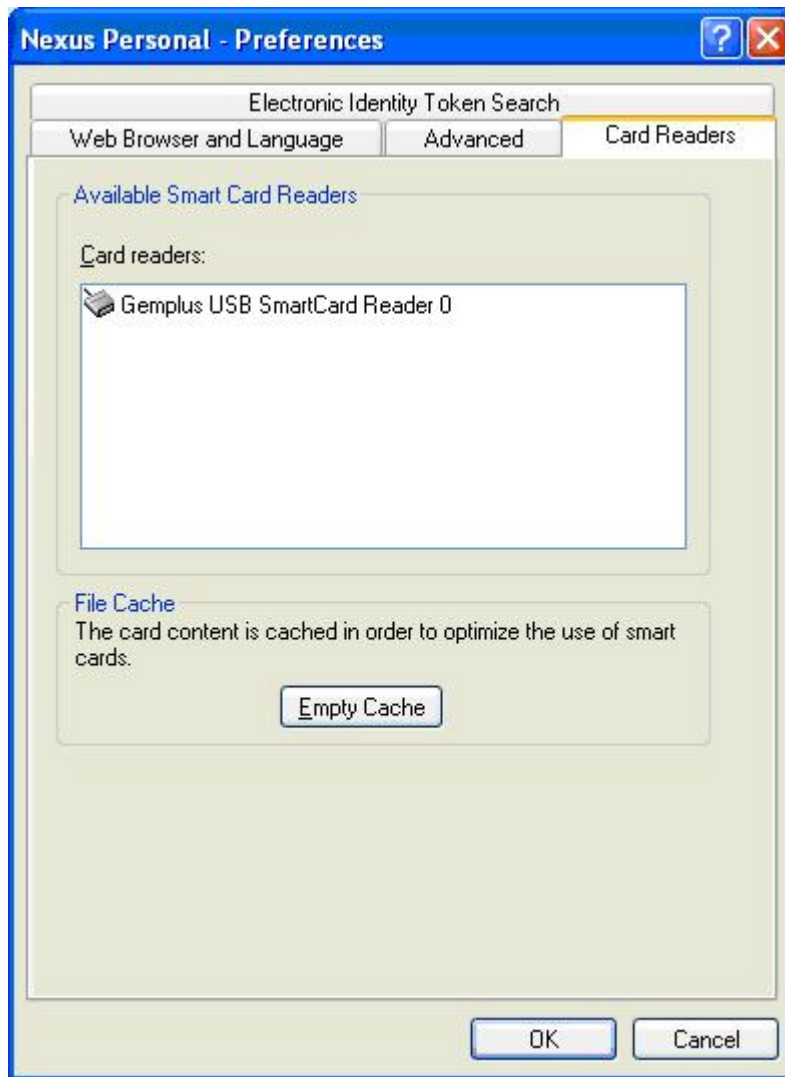
In the Installed Web Browsers entry, information on the installed web browsers and patches is displayed.

Troubleshooting Password Dialogs

In the Troubleshooting Password Dialogs section, settings for secure-desktop issues are found. Here you can activate two methods for disabling the SwitchDesktop command for other processes to ensure secure-desktop functionality and security.

Card Readers

The Card Readers tab contains a list of readers available to Personal.

Figure 15.8. Card Readers

Double-click a reader in the list of available readers to see what version of the reader you have installed.

In order to optimize card access, the contents of a card is cached in Personal. The card serial number is mapped to a particular card file. If the button Empty Cache is clicked, all files containing cached card information will be deleted. This feature is available to prevent a corrupt cache file from making a card unusable. The next time the card is inserted in the card reader, a new cache file will be created.

Tray Icon

Note

This section applies to Windows only.

In the system tray, a Personal tray icon is displayed. By double-clicking on this tray icon, the main application window is launched. When right-clicking on the Personal tray icon, the following options are available:

- By clicking the Tokens command and selecting Enable Token search, the search function described in "Searching for Soft Token" is activated. In addition, all available tokens are shown in the Tokens list. If you click on a token in the list, detailed information about that token will be presented in a window.

- By clicking the Open command, the main application window is launched.
- By clicking Preferences, the main application Preferences window is launched.
- By selecting the Refresh command in the View menu, Personal will be updated with the current contents of the floppy disk. This command is only available if a floppy disk drive is connected to the system and searching for soft tokens on floppy disk is enabled.
- By clicking Exit, Personal will terminate.

About

In the main application window, the command About Nexus Personal can be selected under the Help menu. In the "About Nexus Personal" box, information about Personal is displayed. By clicking the Components... button, all installed Personal components and their versions will be displayed.

Figure 15.9. About



Help

Online help is available from the Help menu. It is also possible to click the F1 button in various dialogs to get context-sensitive help about the active dialog window.

Abbreviations

A

API (Application Programming Interface)

B

BLOB (Binary Large Object)

C

CMC (Common Messaging Call)

COM (Component Object Model) (*Windows only*)

CSP (Cryptographic Service Provider) (*Windows only*)

D

DPAPI (Data Protection API) (*Windows only*)

G

GUI (Graphical User Interface)

GUID () Global Unique Identifier

I

IPC (Inter Process Calls)

M

MSCAPI (Microsoft Cryptographic API) (*Windows only*)

N

NPAPI (Netscape Plugin API)

O

OTP (One-Time Password)

References

- [1] *ANSI X9.17-1995 Financial Institution Key Management (Wholesale)*. . 1995. Appendix C. ¹
- [2] *ANSI X9.31-1998 Digital Signatures using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)* . . 1998. Appendix A.
- [3] *Riemann's hypothesis and tests for primality*. Gary L. Miller. 300–317. 10.1016/S0022-0000(76)80043-8. *Journal of Computer and System Sciences*. 13. 3. December 1976. 0022-0000.
- [4] *Probabilistic algorithm for testing primality*. Michael O. Rabin. 128–138. 10.1016/0022-314X(80)90084-0. *Journal of Number Theory*. 12. 1. February 1980. 0022-314X.
- [5] *Certificate Management Messages over CMS*. M. Myers, , X. Liu, , J. Schaad, , and J. Weinstein. April 2000.
- [6] *PKCS #11 v2.20: Cryptographic Token Interface Standard Draft 4* . .
- [7] *Nexus Personal Message Reference Guide*.
- [8] *Signature Profile for BankID (vers 1.4.3)*.

Note

Technology Nexus AB is not responsible for the contents of external Internet sites.

¹Because ANSI has withdrawn X9.17, the appropriate reference is to ANSI X9.31

Appendix A. Eolas Patent

Introduction

Note

This Appendix only applies to the Windows platform.

As a result of an adverse verdict against Microsoft in a patent infringement lawsuit brought by the University of California and Eolas Technologies, Microsoft may change the way active content is activated in Internet Explorer. This includes loading ActiveX controls, such as the Personal browser plugins.

For the moment, Microsoft has put these changes on hold until questions around validity of the Eolas Patent have been clarified.

More information about the Eolas patent issue can be found on <http://msdn.microsoft.com/ieupdate/>.

The change (if Eolas Patent is valid) will mean that if an ActiveX control is loaded using the <OBJECT> tag and have any <PARAM> tags, the browser will show a warning dialog, and the user will have to press OK, in order to load the control. There are two solutions below describing how to avoid the showing of the warning dialog.

Solution 1

If the control does not load any dynamic data through the <PARAM> tags, such as URIs, it is possible to specify the attribute NOEXTERNALDATA='true' in the <OBJECT> tag, and the warning dialog will not be shown. However if this is done, the browser will not load any <PARAM> that might be URIs. This includes any parameter values including the characters “.” and “/”.

If needed, parameters including these characters (including URIs) might then be set using script functions.

Solution 2

It is also possible not to script the actual control, but rather create a script function that inserts the <OBJECT> and <PARAM> tags into the HTML page. The script function must be defined in a separate file, and not in the HTML file. The browser then should not show the warning dialog.

For more information regarding this workaround, please refer to <http://msdn.microsoft.com/ieupdate/>.

Appendix B. Key Generation

Software Key Generation

Personal uses ANSI X9.17 and a Miller-Rabin test with five repetitions for key generation.

For more information, see the following references: [1], [2], [3], and [4].

Appendix C. Internal Stores

Background

An Internal Store is the place where Personal stores tokens belonging to a user. Each token corresponds to a file in the Internal Store directory. There are two different types of Internal Stores and these are called 1.0 and 1.1 respectively.

Note

The store type is specified per file (as explained in “File Formats” on page 126) and not per Internal Store.

In an Internal Store of type 1.0, the tokens are protected by a PIN code.

In an Internal Store of type 1.1, in addition to the PIN code protection, the tokens are protected by means of a data protection API (DPAPI) that uses encryption with a key derived from the user's Windows logon password (*Windows only*). Personal, or other applications managing PKCS#5, will not be able to read a file protected by DPAPI.

Different Windows versions support different types of Internal Store as indicated by the following tables.

Internal Store types in Personal 4.0

	Win98	WinME	WinXP Home	WinXP Pro	Win2000
All environments	1.0	1.0	1.1	1.1	1.1

Internal Store types in Personal 4.0.1 and later versions

	Win98	WinME	WinXP Home	WinXP Pro	Win2000	Win Vista
Standalone*	1.0	1.0	1.1	1.1	1.1	1.1
Asserted non-NT4 domain**	N/A****	N/A	N/A	1.1	1.1	1.1
Not Asserted non-NT4 domain***	N/A	N/A	N/A	1.0	1.1	1.0

Comments to the table * Computer does not belong to a domain. ** At least one (primary) domain controller responds that it is running either Windows 2000 or Windows 2003. *** "Not asserted non-NT 4 domain" means it cannot be asserted that at least one (primary) domain controller is not running Windows 2000 or Windows 2003. The reason for this can be that the domain controller is not reachable (unplugged network cable) or that it responds that it indeed is running NT4. **** OS cannot belong to a domain.

Note

The type of Internal Store cannot be seen from the application in Personal 4.0 but it is visible in the Token View dialog in Personal 4.0.1 and later.

File Formats

There exist different file formats in different versions of Personal as shown in this table.

File format	Introduced in Personal vers.	Comments
0	4.0	The first version of the file format. Personal will migrate those tokens to the default file format when possible.
1	4.0.1	A version field is introduced in the first ASN.1 object to identify the file version.
2	4.5	In this format, the MAC excludes sub CA certificates.
3	4.7	Encrypted Token objects with PKCS#5 encryption are encoded according to the PKCS#5 v2.0 standard. The difference from format version 2 is that those objects are ASN.1 encoded using the data structures PBES2 and PBKDF2 instead of PBES1 and PBKDF1.
4	4.8	In this format all attributes of a token and its objects are stored in bigendian byte ordering.
5	4.9	In this format, a bug, introduced in Personal 4.8, has been fixed. With this format it is now possible to move a token between format 1.0 and 1.1 without having problems with PIN verification.

Note

The file format of the stored tokens cannot be seen from the GUI in either version of the application.

The file format used in Personal 4.0 is not supported in later versions of Personal, but it is possible to convert the file format of the Internal Store tokens created in Personal 4.0 if the Migration Wizard is invoked.

If there are tokens in file format version 0 when a later version of Personal is installed, the Migration Wizard will be started automatically and a PIN code must be entered for each token to be migrated. The new file format will vary depending on which version of Personal is installed.

If there are tokens in file format versions 0, 1 or 2 when Personal 4.7 or later is installed, these tokens will be upgraded to version 3 automatically in any scenario where the user needs to enter the PIN code, since the PIN code is required in order for the format upgrade to take place.

Object Identifiers related to the Internal Store Token

```
ProtectedStoreToken OBJECT IDENTIFIER ::= 1.2.752.36.4.1.2
P5EncryptedObject OBJECT IDENTIFIER ::= 1.2.752.36.4.1.3
DPAPIEncryptedObject OBJECT IDENTIFIER ::= 1.2.752.36.4.1.4
DPAPIEncryptionAlgorithmId OBJECT IDENTIFIER ::= 1.2.752.36.4.1.5
```

A token file in Personal 4.0.1 and later has the following ASN.1 structure.

```
Protected store token ::= SEQUENCE {
  fileVersion FileVersion,
  attributes Attributes,
  softTokenObjects SoftTokenObjects,
  mac Mac
}
FileVersion ::= INTEGER
Attributes ::= OCTET STRING
SoftTokenObjects ::= SET OF SoftTokenObject
SoftTokenObject ::= SEQUENCE {
  identifier OBJECT IDENTIFIER
  objectData OCTET STRING
}
When the SoftTokenObject is a ProtectedStoreObject (1.2.752.36.4.1.2) the
objectData OCTET STRING encapsulates
SEQUENCE {
  attributes OCTET STRING
  objectData OCTET STRING
}
Mac ::= OCTET STRING
```

In this file format, the first ASN.1 object contains an integer to identify the file format version. The value for a specific version of Personal can be read from the table in “File Formats” on page 126.

In addition, the MAC value is calculated differently (see “MAC Calculation” on page 131 for more information).

The private key storage without DPAPI protection has the following structure in file format versions 0, 1 and 2.

```
OBJECT IDENTIFIER '1 2 752 36 4 1 3' (P5EncryptedObject)
OCTET STRING, encapsulates {
  SEQUENCE {
    OBJECT IDENTIFIER '1 2 752 36 4 1 2' (ProtectedStoreToken)
    OCTET STRING, encapsulates {
      SEQUENCE {
        OCTET STRING
        [00 00 00 00 00 00 00 04 ...]
        OCTET STRING, encapsulates {
          SEQUENCE {
            INTEGER 0
            SEQUENCE {
              OBJECT IDENTIFIER
              data (1 2 840 113549 1 7 1) (PKCS #7)
            }
            SEQUENCE {
              OBJECT IDENTIFIER
              pbeWithSHAAnd3-KeyTripleDES-CBC (1 2 840 113549 1 12 1 3) (PKCS #12 PbeIds)
            }
            SEQUENCE {
              OCTET STRING
              [D6 C0 2C 96 AD 3A 3D 4D B2 FE...]
              INTEGER 8192
            }
          }
        }
      }
    }
  }
}
```



```

OBJECT IDENTIFIER '1 2 752 36 4 1 3' (P5EncryptedObject)
OCTET STRING, encapsulates {
  SEQUENCE {
    OBJECT IDENTIFIER '1 2 752 36 4 1 4' (DPAPIEncryptedObject)
    OCTET STRING, encapsulates {
      SEQUENCE {
        OBJECT IDENTIFIER '1 2 752 36 4 1 2' (ProtectedStoreToken)
        OCTET STRING, encapsulates {
          SEQUENCE {
            OCTET STRING [00 00 00 00 00 00 00 00 ...]
            OCTET STRING, encapsulates {
              SEQUENCE {
                INTEGER 0
                SEQUENCE {
                  OBJECT IDENTIFIER data (1 2 840 113549 1 7 1) (PKCS #7)
                  SEQUENCE {
                    OBJECT IDENTIFIER '1 2 752 36 4 1 5'
                      (DPAPIEncryptionAlgorithmId)
                    NULL
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

```

(**) is the data that when decrypted with DPAPI will yield the underlying PKCS#5 encrypted private key as shown in one of the two previous examples, depending on the file format version.

MAC Calculation

The MAC calculation varies depending on the version of the file format.

- For version 1, the MAC includes all CA certificates on the token.
- For version 2 and later, the MAC includes only the Root CA.

Appendix D. CSP and PKCS#11 Configuring

Platform dependencies

Note

CSP only applies to the Windows platform.

Implementation

In the configuration file `personal.cfg` (on Windows and Linux) and `se.nexus.personal.cfg` (on Mac), there is a section named `[CSP_PKCS11]` holding settings related to PKCS#11 and CSP.

In this section there is a parameter named `Sections` defining which applications are using specific settings. If there is more than one application, they should be separated by semicolons.

CSP is using the config API to get settings from configuration file. The config API works according to the following principles:

1. Get the process name and check if it is one of the applications specified by parameter `Sections`.
2. When the name is specified by `Sections` and there is a key defined for that process, use that key. If a key is missing use the value in `[CSP_PKCS11]` containing the global settings.
3. If the specified config parameter is neither in the process specific section nor in the global section, the application will behave according to the default settings compiled into CSP and PKCS#11.

When CSP and PKCS#11 are loaded by an application, the config file is automatically updated with a section for the calling application if that section does not already exist. This means that if, for example, the program `CTest` (`CTest.exe`) is loading PKCS#11, the first time, the config file will be updated. `CTest.exe` is added to the parameter `Sections` and a new section `[ctest.exe]` is created for `CTest`.

Example: The following example will show a typical excerpt from a config file:

```
[CSP_PKCS11]
CSP_DefaultKeyContainer=\\.\Card Reader X 0\
CSP_IgnoreFlagSilent=0
P11_AlwaysLoggedInMode=0
Sections=app1.exe;app2.exe
[app1.exe]
CSP_EnableFlagNoHashOid=0
CSP_IgnoreFlagSilent=1
[app2.exe]
P11_AlwaysLoggedInMode=1
CSP_DefaultKeyContainer=\\.\Card Reader 2 0\
```

Appendix E. PIN-Related Issues

PIN Caching

In Personal it is possible to cache PIN codes in the PKCS#11 and the CSP (*Windows only*) modules. This is the case when Personal is run in PIN caching mode. Personal can also run in PIN non-caching mode. Then no PIN caching will take place.

The PIN caching can be configured with the configuration parameters `CSP_EnableCachePIN` and `P11_EnableCachePIN` in `personal.cfg`. If set to 1, PIN caching is enabled, and when set to 0, PIN caching is disabled.

PIN caching applies to all tokens, i.e. both smart card and software tokens.

This appendix highlights various aspects of the different operation modes.

Note

When the term card is used throughout this description it only applies to smart cards. When the term token is used it applies to both smart cards and software tokens.

Sometimes, card specific information is marked with the icon



PIN Caching Mode

Each process that loads the PKCS#11 and the CSP modules has its own PIN caching environment. When a successful login is performed, the PIN is internally logged out and is stored in the PIN caching environment in the given process. The cached PIN will later be used as soon as a private operation is performed. As soon as a private operation is performed the PIN is internally logged out.

If the CSP and PKCS#11 are loaded into the same process, they will share PIN caching environment.



When running an application in PIN caching mode, the card is always accessible from other applications since the card will always be released when an operation that needs private card access has been performed.

Note

This section only applies to the Windows platform.

A PIN is cached on a token basis. In one process, all contexts accessing a specific token will have access to the cached PIN code. The PIN code is cleared from the cache as soon as the last context of a specific token is released, or when the token is removed from its reader.

PKCS#11 Specific Information

Since every PKCS#11 slot represents one PIN, the PIN is cached on a slot basis. In one process, all sessions accessing a specific slot will have access to the cached PIN code. As soon as `C_Logout` is called, or the last session to a specific slot is closed, the PIN is cleared from the cache. Also, if the specific token is removed, the PIN is cleared from the cache.

PIN Non-Caching Mode

In PIN non-caching mode the calling application has the control of the login states of the tokens.

If the CSP and PKCS#11 are loaded into the same process, they will share the login state of the token.



If no PIN caching is used, the card will be exclusively locked by the calling process as long as the user is logged in. Thus, it will not be accessible from other applications until it is released.

CSP Specific Information

Note

This section only applies to the Windows platform.

If a context is logged into a token, that logged in state can be used by all other contexts accessing that specific token. The token will be logged out as soon as the last context to the token is released.

A card will not be accessible to other processes until the card is logged out.

It is possible to configure the CSP to always log out after it has performed an operation that needs logged in state. The configuration parameter is `CSP_LogoutAfterSign` and when set to 1, the token is logged out after a private operation.

PKCS#11 Specific Information

If a session is logged into a slot, the slot will be accessible from all other sessions to that slot in the calling process. The token will be logged out as soon as `C_Logout` is called, or all sessions to a given slot are closed.

A card will not be accessible to other processes until the card is logged out.

It is possible to configure the PKCS#11 to always log out after it has performed a signing operation. The configuration parameter is `P11_LogoutAfterSign` and when set to 1, the token is logged out after a signing operation.

Configuration Details

Assume that we have a scenario where the user has logged into a token in order to sign a hash with his private key present on the token.

The table below, which applies to the Windows platform only, shows the behavior of the CSP in different combinations of the flags `CSP_EnableCachePIN` and `CSP_LogoutAfterSign`.

	CSP_EnableCachePIN=0	CSP_EnableCachePIN=1(default)
CSP_LogoutAfterSign=0(default)	A card that is logged in is locked by the calling application until the last context to the card is released.	A card is released after every private operation and will thus be accessible to other processes.
CSP_LogoutAfterSign=1	A logout is forced after every private operation. As a result, the user will be asked for his PIN prior to every operation requiring that he is logged in.	A logout is forced after every private operation. As a result, the user will be asked for his PIN prior to every operation requiring that he is logged in.

The table below shows the behavior of the PKCS#11 in different combinations of the flags `P11_EnableCachePIN` and `P11_LogoutAfterSign`. `P11_EnableCachePIN=0`

	P11_EnableCachePIN=1(default)	P11_LogoutAfterSign=0(default)
P11_LogoutAfterSign=0(default)	A card will not be accessible from other applications as long as it is in logged in state.	The token is internally released after every private operation. A card will be accessible from

	P11_EnableCachePIN=1(default)	P11_LogoutAfterSign=0(default)
		other applications even if it is logged in.
P11_LogoutAfterSign=1	A logout is forced after every signing operation. A card will not be accessible from other applications as long as it is in logged in state.	A logout is forced after every signing operation. The token is internally released after every private operation. A card will be accessible from other applications even if it is logged in.

Force Login Before Sign

Personal can be configured to require a PIN-reverification prior to a signing operation with a key, when the key or the corresponding certificate has a specific key usage.

The configuration parameter is `ForceLoginBeforeSignKeyUsage` in the file `personal.cfg`. (See the delivered sample configuration file for more information about this parameter.)

Only the key usage extension Non-Repudiation is supported.

Example: If PIN-reverification should be performed prior to every non-repudiation signature, Personal should be configured as follows:

```
[CSP_PKCS11]
ForceLoginBeforeSignKeyUsage=0x40
```