

RSA ACE/Agent Authentication API 5.0 Guide

Adding RSA SecurID Protection to Your Software Applications

RSA Security Inc.
20 Crosby Drive
Bedford, MA 01730-1402

Main number: 781.687.7000
Customer Services: 800.995.5095
International calls: Refer to Web site for specific countries.
e-Mail: info@rsasecurity.com
Web Site: www.rsasecurity.com

See our Web Site for regional Customer Service telephone and fax numbers.

Trademarks

ACE/Server, BSAFE, Keon, RC2, RC4, RSA Data Security, Inc., The Keys to Privacy and Authentication, RSA SecurPC, SecurCare, SecurID, Security Dynamics, SoftID, and WebID are registered trademarks, and ACE/Sentry, BCERT, Genuine RSA Encryption Engine, JSAFE, RC5, RC6, RSA, RSA Secured, SecurSight, and The Most Trusted Name in e-Security are trademarks, of RSA Security Inc.

Other product and company names mentioned herein may be the trademarks of their respective owners.

License agreement

This software and the associated documentation are proprietary and confidential to RSA Security Inc., are furnished under license, and may be used and copied only in accordance with the terms of such license and with the inclusion of the copyright below. This software and any copies thereof may not be provided or otherwise made available to any other person. No title to or ownership of the software and associated documentation is hereby transferred. Any unauthorized use or reproduction of this software or documentation may be subject to civil or criminal liability. The information in the software and documentation is subject to change and should not be construed as a commitment by RSA Security Inc.

Restricted rights notice for license to the U.S. Government.

Use, reproduction, or disclosure is subject to restrictions as stated in the "Rights in Technical Data—General" clause (alternate III), in FAR section 52.222-14. All title and ownership in this computer software remain with RSA Security Inc.

Note on DES and other encryption

This product contains the DES (data-encryption standard) algorithm and RSA Security's own proprietary encryption method. Exporting these encryption algorithms to certain countries may be prohibited or restricted by the laws of the United States. Current U.S. export regulations should be followed when exporting this product outside the United States.

Distribution

Limit distribution of this document to trusted personnel.

RSA notice

The RC5™ Block Encryption Algorithm With Data-Dependent Rotations is protected by U.S. Patent #5,724,428 and #5,835,600.

Contents

Preface	5
Developer Profile	5
Available Resources	5
Related Documentation	5
Sample Source Code	6
Getting Support and Service	6
What's New in Version 5.0	7
Backwards Compatibility	7
Changes and Additions to the API	7
New Two-Step Authentication Support	7
New Load Balancing and Failover	8
Thread Safety	8
Data Encapsulation	8
Using AceInitialize in the Windows NT Environment	9
Using Synchronous APIs with Authentication Context Data	9
Checking the Status of Asynchronous Functions	9
Expiration Facility and Cleanup Callbacks	10
Message Catalog for Prompts and Log Messages	10
UNIX Library Filename Change	10
List of New and Updated Functions	10
RSA ACE/Agent Authentication API Overview	13
Two-Factor Authentication	13
The Role of an RSA ACE/Agent	14
Using the RSA ACE/Agent Authentication API	17
Selecting the Correct Function	17
API Taxonomy	18
Synchronous and Asynchronous Functions	20
Compiling and Linking with the RSA ACE/Agent Authentication API 5.0	20
Compiling on UNIX and Windows NT	20
Using Dynamic Libraries or Static Libraries in UNIX	21
API Functions	23
AceCancelPin	24
AceCheck	26
AceCleanup	28
AceClientCheck	29
AceClose	31
AceCloseAuth	33
AceContinueAuth	34
AceGetAlphanumeric	36

AceGetAuthenticationStatus.....	37
AceGetMaxPinLen	38
AceGetMinPinLen.....	39
AceGetPinParams.....	40
AceGetShell.....	41
AceGetSystemPin.....	42
AceGetTime	43
AceGetUserData.....	44
AceGetUserSelectable.....	45
AceInit.....	46
AceInitialize	48
AceLock	49
AceSendNextPasscode	51
AceSendPin	53
AceSetNextPasscode	55
AceSetPasscode.....	56
AceSetPin	57
AceSetTimeout	58
AceSetUserClientAddress	59
AceSetUserData.....	60
AceSetUsername	61
AceShutdown	62
AceStartAuth	63
SD_Check.....	65
SD_ClientCheck.....	66
SD_Close.....	67
SD_Init	68
SD_Lock.....	69
SD_Next	70
SD_Pin.....	71
Appendix A: Return Values and Result Codes	73
Return Values	73
Result Codes	74
ACM_OK	75
Appendix B: Using Synchronous and Asynchronous API Functions	77
Appendix C: Modifying the Message Catalog	79
Appendix D: Legacy Functions	81
Index.....	83

Preface

The RSA ACE/Agent Authentication Application Programming Interface (API) enables developers to integrate RSA SecurID® into custom or third-party applications. The *RSA ACE/Agent Authentication API 5.0 Guide* contains the latest information about the RSA ACE/Agent Authentication API functions.

The “What’s New in Version 5.0” section provides important information concerning features of version 5.0 of the API. It will be most relevant to developers who have prior experience using the API, but newcomers will also find it informative. The “RSA ACE/Agent Authentication API Overview” section is a high-level description of the processing required by any Agent that is to successfully authenticate to an RSA ACE/Server®. The “Using the RSA ACE/Agent Authentication API” section contains information to help developers select the right API functions for particular tasks and important instructions for compiling and linking code that uses the API on either UNIX (Solaris, HP-UX, or AIX) or Windows NT. The “API Functions” section contains detailed information on all functions available in version 5.0 of the API, with cross-references to other parts of this guide for additional information.

Appendix A contains lists of all return values and result codes, as well as the typical meaning of each one. Appendix B is a table that describes when synchronous and asynchronous API function calls can be used with one another. Appendix C describes how to modify the message catalog containing strings that can be localized for non-English speaking sites. Finally, Appendix D provides a list of legacy API functions for which support is being phased out.

Developer Profile

It is very helpful, but not necessary, to have previous experience in RSA ACE/Agent development. You should have an understanding of both synchronous and asynchronous function calls, callback mechanisms, and the implications of designing and developing multithreaded code

Available Resources

Related Documentation

RSA ACE/Agents ship with documentation that describes how to install, configure, use, and troubleshoot the Agent on its designated platform. RSA ACE/Server ships with a documentation set that describes how to install, configure, deploy, and troubleshoot RSA ACE/Servers.

About Printing PDF Files

Acrobat Reader is available from www.adobe.com. If you are using Adobe Acrobat Reader v 4.0 to print a PDF document, deselect the **Fit to Page** option in the Print dialog box for the document to print at the correct size.

Sample Source Code

Sample source code that demonstrates the basic calling sequence and usage of both the synchronous and asynchronous API function calls is included on the distribution CD-ROM, as follows:

\ACEAgentSDK\samples\async. This sample demonstrates the asynchronous API calls. Although the sample illustrates the calling mechanisms that are required when using the asynchronous API, it does not do so in the context of a truly asynchronous application.

\ACEAgentSDK\samples\sync. This sample demonstrates use of the synchronous API calls **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth**. The calls are provided to simplify the processing of RSA SecurID authentication.

\ACEAgentSDK\samples\sync2. This sample demonstrates use of the new thread-safe synchronous API calls **SD_Close**, **SD_Init**, **SD_Next**, and **SD_Pin**. These calls are the recommended mechanism for porting existing “RSA SecurID Ready” products to the new API library.

\ACEAgentSDK\samples\sync3. This sample is the original **example2** sample program from previous releases of the API, now ported to use the same synchronous API calls as those used by the **sync2** sample. The **sync3** sample demonstrates how to upgrade code that uses pre-version 5.0 calls.

If you will use the API calls in a Windows NT environment, look at the two synchronous code sample files **sync** and **sync2** in the **\ACEAgentSDK\samples** directory. You can also find the asynchronous code sample file **async** in the **\ACEAgentSDK\samples** directory. If you will use the API calls in a UNIX (Solaris, HP-UX, or AIX) environment, the API functions can be found in the **ACEAgentSDK/lib** directory on the distribution CD-ROM.

Note: RSA Security recommends that you download the most recent updates to the sample source code files from the RSA Security SecurCare Online Web site. To download the latest RSA ACE/Agent Authentication API sample code, visit www.rsasecurity.com/support/securcare.

Getting Support and Service

SecurCare® Online	www.rsasecurity.com/support/securcare
General Technical Support Information	www.rsasecurity.com/support
Technical Support Telephone Numbers	www.rsasecurity.com/support/news/tollfree.html
Call Handling and Escalation Process	www.rsasecurity.com/support/news/escproc.html

Important: There is no provision for technical support unless valid RSA Security Maintenance and Developer Support contracts are in effect. For more information, contact your RSA Security Sales Representative or Partner Marketing Manager.

What's New in Version 5.0

This section describes the features of the RSA ACE/Agent Authentication Application Programming Interface (API) that are new, or have changed in important ways, since the release of previous versions of the API. Although this section is most relevant for developers who have experience with prior versions of the API, the information presented will also help developers who are new to the API understand why and how some of the API functions should be used.

Backwards Compatibility

Version 5.0 of the Authentication API will interoperate with all currently supported versions of the RSA ACE/Server, including versions 3.3.1, 4.0, 4.1, and 5.0. Furthermore, version 5.0 fully supports all of the functions present in version 4.x of the API.

As version 5.0 RSA ACE/Servers are deployed, older Agents can take advantage of the new API load-balancing capabilities and faster encryption routines by relinking against version 5.0 of the API. No other changes to existing Agents are required. RSA Security *strongly* recommends, however, that old Agent source code be updated to use the new calls for two-step authentication and multithreaded application support. In addition, applications for the RSA Security Secured Partners certification program will not be accepted if the corresponding code fails to support two-step authentication.

Changes and Additions to the API

New Two-Step Authentication Support

Prior to the release of version 5.0, an Agent was restricted to interacting with two RSA ACE/Servers: a Master Server and a Slave Server. Only one Server at a time was permitted to authenticate users for the Agent. With the release of version 5.0, multiple Servers within the same RSA ACE/Server realm can authenticate users for an Agent.

To avoid the possibility of an unauthorized person capturing information from an RSA SecurID authentication request and using it to authenticate to another Server, the Agent first sends a lock request to a Server before sending the user's PASSCODE. The Server locks the username, which prevents it from being accepted by any other Server in the realm, and waits for the corresponding PASSCODE to be submitted. After the user sends the correct PASSCODE, the Server unlocks the username and the user is authenticated. This new process is referred to as two-step authentication and is supported by the new functions **AceLock** and **SD_Lock**.

For more information about the **AceLock** and **SD_Lock** functions, refer to their descriptions in this guide. For more information about what an administrator must do to support two-step authentication on the Servers, refer to the RSA ACE/Server 5.0 documentation.

Note: If you use version 5.0 of the RSA ACE/Agent Authentication API with existing Agents, you do not have to change your use of the **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth** functions to support two-step authentication. Two-step authentication is automatically performed within these three functions.

New Load Balancing and Failover

An Agent that relinks against version 5.0 of the Authentication API can now communicate with multiple RSA ACE/Servers in a realm. Version 5 of the API incorporates load-balancing routines to help the Agent select the best Server with which to communicate. Load balancing will occur automatically based on a combination of data from the mandatory configuration file **sdconf.rec**, the optional configuration file **sdopts.rec**, and data determined dynamically by the Agent at run time. Agent developers need take no special actions to invoke load balancing or failover functionality.

In addition to affecting load balancing, **sdconf.rec** and **sdopts.rec** can provide both real and alias addresses for Servers in the realm. The **sdopts.rec** file can also provide an overriding IP address for an Agent. These configuration files are described in detail in the RSA ACE/Server 5.0 documentation.

Thread Safety

All API functions described in this document are thread safe. That is, they can safely be called from multithreaded applications without program failure or data corruption.

Note: The API **sd_XXX** function calls listed in Appendix D of this guide are still supported but are not thread safe. RSA Security *strongly* recommends that you replace any existing **sd_XXX** calls with the new **SD_XXX** calls to take advantage of two-step authentication and multithreaded application support. In future versions of the API, the **sd_XXX** functions will *not* be supported.

Data Encapsulation

In previous versions of the API, a developer could directly access the contents of API-specific data structures. In version 5.0, you cannot access data directly in the **SD_CLIENT** data structure. Instead of direct access, your code must obtain a handle to the data area, which should then be used in API functions that provide the content of specific elements within the data structure.

The following functions will provide data that was previously available by directly reading API data structures: **AceGetAlphanumeric**, **AceGetAuthenticationStatus**, **AceGetMaxPinLen**, **AceGetMinPinLen**, **AceGetShell**, **AceGetSystemPin**, **AceGetTime**, **AceGetUserData**, and **AceGetUserSelectable**.

Using AceInitialize in the Windows NT Environment

In earlier versions of the Authentication API, calls to **AceInitialize** were necessary only for the UNIX library to initialize the threads on UNIX and to read data from the RSA ACE/Server configuration file, **sdconf.rec**. Now your code must also call **AceInitialize** in the Windows NT environment. RSA Security *requires* that your code always call **AceInitialize** once, before calling any other API function.

It is your code's responsibility to call **AceInitialize**. To help avoid a potential error condition, **AceInit**, **AceStartAuth**, **SD_Init**, and **sd_init** will detect whether the **AceInitialize** function has already been called, and will call the function for your code if necessary. However, if your application calls **AceInit**, **AceStartAuth**, or **SD_Init** in a multithreaded environment, and multiple automatic calls to **AceInitialize** happen at the same time, those functions will return an error. In that case, your code must make a call to **AceInitialize** once *before* calling **AceInit**, **AceStartAuth**, or **SD_Init**.

Using Synchronous APIs with Authentication Context Data

Two sets of synchronous API function calls are now available. One set, consisting of the **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth** calls, has been enhanced by allowing for any authentication-specific context data to be stored in the user data context. A new function, **AceSetUserData**, allows users of these three synchronous calls to store information in the form of a 32-bit value (it can be a pointer to data), which can be retrieved later by the **AceGetUserData** call. Uses of **AceSetUserData** and **AceGetUserData** are included in the detailed description of those functions.

The other synchronous API set includes the calls **SD_Init**, **SD_Check**, **SD_Pin**, **SD_Next**, and **SD_Close**. These five calls closely mirror the legacy calls **sd_init**, **sd_check**, **sd_pin**, **sd_next**, and **sd_close**, but the newer calls are thread safe and can be used with the newer context data functions.

Checking the Status of Asynchronous Functions

To use the asynchronous API, you should know how to use callback mechanisms.

Your code can make a call to an asynchronous function, but should not proceed to the next step in the authentication process until the function has completed successfully. Of course, you can include code that performs other functions while waiting for an asynchronous function to finish.

All asynchronous calls return quickly with a status that indicates success or failure of the function call, *not* whether the user successfully authenticated. Your code should check this status, and you should be aware that before a successful return from an asynchronous function occurs, other threads have been enabled. A developer-supplied callback function is called when these threads finish. It is only upon the successful completion of all the threads associated with an asynchronous authentication call that the status of the authentication request can be determined.

As part of the callback method, your code can call **AceGetAuthenticationStatus** to find out the status of a specific authentication request. Refer to the note on page 23 and the description of **AceGetAuthenticationStatus** on page 37 for more information.

Important: There is a design constraint on the use of **SDI_HANDLE**. An asynchronous application of the API *must* avoid concurrent use of the *same* **SDI_HANDLE**. Multiple threads using the same **SDI_HANDLE** will return incorrect results.

Expiration Facility and Cleanup Callbacks

The API now has a facility that allows any outstanding authentication request to be terminated automatically. The **AceSetTimeout** function call implements this facility. In addition to this call, the calls **AceCleanup** and **AceShutdown** are used to gracefully terminate outstanding authentication requests and to cleanly shut down the library.

Message Catalog for Prompts and Log Messages

Prior versions of the Authentication API included hardcoded English strings that were returned by **AceStartAuth** and **AceContinueAuth**. These strings have been moved to a message catalog that you can customize. The same catalog contains all of the messages the API writes to the UNIX syslog and Windows NT Event Log. See “Appendix C: Modifying the Message Catalog” for information on how to manipulate the message catalog to change the prompts or translate the strings into other languages.

UNIX Library Filename Change

In previous versions of the RSA ACE/Agent Authentication API for UNIX platforms, the API library was named **sdiclient.a**. In version 5.0, the library is now named **libaceclnt.a**. A shared version of this library is also available on the distribution CD-ROM.

List of New and Updated Functions

The following functions have been added or updated since previous releases of the RSA ACE/Agent Authentication API:

AceCleanup

This call will terminate all outstanding authentication requests. Any requests in progress will return with **ACCESS_DENIED**.

AceGetPinParams

This new call obtains all of the PIN-related parameters at once.

AceInitialize

In prior versions of the API, this call was only used in UNIX Agents. Now all Agents, including those that operate in Windows NT, must call this function once before any other call is made.

AceLock

This new call implements the lock request for two-step authentication.

AceSetTimeout

This call was added to allow for a timed expiration of outstanding authentication requests.

AceSetUserData

This function was added both to allow your code to supercede the user data specified during the **AceInit** call with new data, and to give the synchronous API calls the same overall functionality as **AceInit**.

AceShutdown

This call complements the **AceInitialize** call. Your code should call this function if it becomes necessary to unload a dynamically linked RSA ACE/Agent API library. It is necessary to properly terminate the internal threads and perform cleanup before the library is unloaded.

SD_Check

This call is a synchronous wrapper for the functions **AceCheck**, **AceSetUsername**, and **AceSetPasscode**. It supercedes **sd_check**.

SD_ClientCheck

This new call is a synchronous wrapper for the **AceClientCheck**, **AceSetUsername**, **AceSetPasscode**, and **AceSetUserClientAddress** functions.

SD_Close

This call is a synchronous wrapper for **AceClose**. It supercedes **sd_close**.

SD_Init

This call is a synchronous wrapper for **AceInit**. It supercedes **sd_init**.

SD_Lock

This new call is a synchronous wrapper for the **AceSetUsername** and **AceLock** functions.

SD_Next

This call is a synchronous wrapper for the **AceSetNextPasscode** and **AceSendNextPasscode** functions. It supercedes **sd_next**.

SD_Pin

This call is a synchronous wrapper for the **AceSetPin**, **AceSendPin**, and **AceCancelPin** functions. It supercedes **sd_pin**.

RSA ACE/Agent Authentication API Overview

Two-Factor Authentication

The RSA Security solution for strong user authentication – RSA SecurID – is based on an approach called *two-factor authentication*. The premise of this approach is that a single, remembered factor, such as a password, inherently provides a low proof of authenticity, because anyone who overhears or steals the password will appear completely genuine. The addition of a second, physical proof makes the certainty of authenticity considerably higher.

With RSA SecurID, authorized users are issued individually registered authentication devices, or RSA SecurID *tokens*, that generate single-use numeric codes known as *tokencodes*. Tokencodes change based on a time code algorithm. Every 60 seconds a different tokencode is generated. An RSA ACE/Server protects the network by validating the changing code. Each token is unique and it is impossible to predict the value of a future tokencode by recording prior tokencodes. The combination of a secret password and a valid tokencode provides the two factors necessary to ensure strong authentication.

The Role of an RSA ACE/Agent

User authentication for local network access, remote dial-in, Internet/VPN connections, intranet/extranet applications, or customized network services is accomplished with an RSA ACE/Server, working in conjunction with an RSA ACE/Agent. Figure 1 shows a deployment of RSA ACE/Agents in a typical network environment, consisting of Internet and intranet access separated by a network *demilitarized zone* (DMZ).

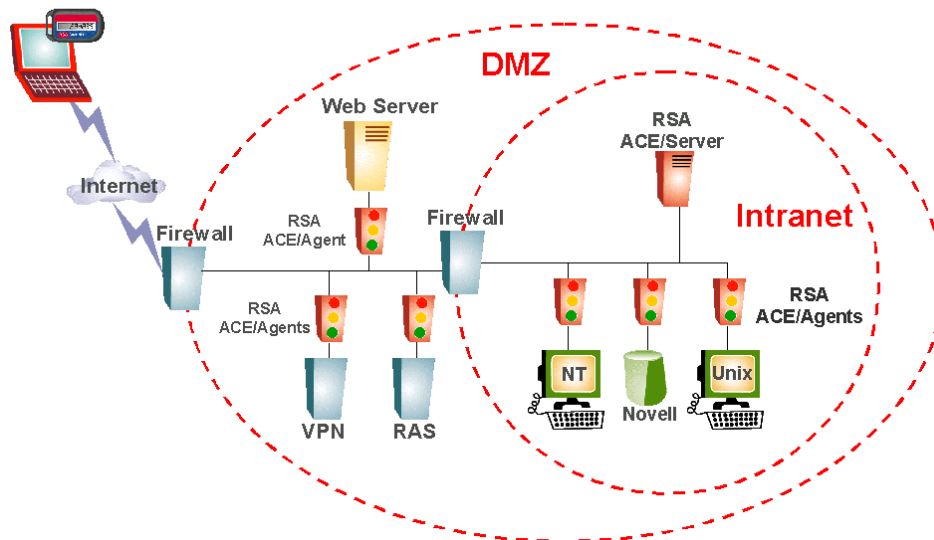


Figure 1: A Typical Network Deployment of RSA ACE/Agent Components

Agents are software components that initiate an RSA ACE/Server authentication session when a user attempts to access a protected resource. RSA Security offers a number of Agents that are designed to protect operating systems and data hosted on Web servers. In addition, a long list of third parties have embedded Agents in their products (for example, in remote access servers, firewalls, and routers). Custom Agents, created using version 5.0 of the RSA ACE/Agent Authentication API, can be developed to protect a myriad of resources, and can offer many end user and administrator capabilities.

Agents perform the following steps in a secure manner:

1. Intercept all access attempts of a particular type (such as attempts to log in or access a URL).
2. Determine whether the specific requested resource is protected by RSA SecurID.
 - If the requested resource is not protected, the Agent either ignores the request or, in the case of a custom Agent, takes whatever action is appropriate (such as writing an audit message in the UNIX syslog or the Windows NT Event Log).
 - If the requested resource *is* protected by RSA SecurID, then processing should continue.

3. Determine the username of the individual requesting access to the protected resource, so the RSA ACE/Server can validate that the tokencode received later in the process comes from the authentication device registered to the requesting user. Usernames can be entered by users in response to a prompt, or the names can be determined by consulting system parameters specific to the particular environment in which the Agent is executing.
4. In an environment using the RSA ACE/Server 5.0 replication capabilities, lock the username to prevent replay attacks.
5. Request the user's PASSCODE, which consists of the Personal Identification Number (PIN) that is known only by the user whose name has been supplied (one authentication factor), and the tokencode currently displayed by the user's token (the second authentication factor).
6. Combine the PASSCODE with data known only to the Agent and its associated RSA ACE/Servers in the realm and deliver the combined data to a Server for validation.
 - If an RSA ACE/Server approves the access request, the Agent should permit access to the requested resource and take whatever other Agent-specific actions might be appropriate (for example, logging a user on to a system).
 - If a Server denies the access request, the Agent must prevent the user from getting access to the protected resource, and takes whatever other actions that might be appropriate (for example, writing an audit message in the UNIX syslog or the Windows NT Event Log).

In addition to providing basic access checks, an Agent should also handle several security-related housekeeping tasks. In particular, some of the security and usability features of RSA SecurID require that Agents properly handle situations where a user must set a PIN, and situations when additional tokencodes are required to foil potential security attacks.

Using the RSA ACE/Agent Authentication API

A critical feature of an Agent is its ability to securely implement the RSA SecurID protocol and communicate with an RSA ACE/Server to process user authentications. The particulars of the RSA SecurID protocol and the knowledge needed to successfully communicate with a Server have been encapsulated in the functions that comprise the RSA ACE/Agent Authentication API. Developers may use this Authentication API to construct custom Agents that satisfy specific mission-critical needs within an organization.

Important: The specifics of using the various functions described in this manual are demonstrated in sample programs that are provided both on the distribution CD-ROM, and on the www.rsasecurity.com/support/securcare Web site. You should closely examine the source code for each of the samples to understand the proper calling sequences required. Each sample demonstrates a different set of functions.

Selecting the Correct Function

When implementing a custom Agent, your code must correctly handle several cases that might arise during execution:

“Normal” authentication. A user enters a username and PASSCODE for validation by an RSA ACE/Server. Note that the return values from many of the API functions do not indicate whether any particular authentication was successful. The only way to determine if an authentication was successful in some cases is by testing the result code, using the function **AceGetAuthenticationStatus**.

Next Tokencode Mode. The RSA ACE/Server has requested the next tokencode displayed on the user’s token. There are several security-relevant situations in which this case might occur. If the next tokencode is not properly sent to the RSA ACE/Server, the corresponding authentication will fail.

New PIN Mode. The RSA ACE/Server administrator has determined that the user associated with a particular token must have a new PIN. The RSA ACE/Server administrator determines the characteristics of PINs (for example, minimum and maximum length, whether numeric or alphanumeric, and whether user-selectable or system-generated), which can be tested using API functions to assist in I/O.

With two exceptions, the Agent, as opposed to the Authentication API library, is responsible for all input and output associated with processing an authentication request. That is, the Agent must prompt the user with strings designed by the Agent developer (for example, “Enter PASSCODE:”), and the Agent must display status information as determined by the developer based on the return values and result codes (for example, “Access denied.”). The only exceptions are the functions **AceStartAuth** and **AceContinueAuth**, which will supply prompts that the Agent should display to the user. Refer to “Appendix C: Modifying the Message Catalog” for additional information on these prompts.

Different functions will be used at different times, depending on the case being handled and whether the Agent is synchronous or asynchronous. The following sections provide additional information about how to use the available routines.

API Taxonomy

Conceptually, the API functions can be grouped into five categories:

Authentication. These functions are directly involved in authenticating a user’s identity, mostly by setting values in data structures that will be sent to the RSA ACE/Server for validation. The data structures cannot be accessed directly, but rather are referenced by a handle that is given a value (using “Housekeeping” functions) during the initialization process for a particular authentication sequence.

PIN Processing. These functions return characteristics of acceptable PINs and process user PINs using the RSA ACE/Server. Knowing the characteristics of valid PINS (for example, minimum and maximum length, whether numeric or alphanumeric, whether user selectable or system-generated) will help with Agent I/O during New PIN Mode. The PIN processing functions are needed to handle New PIN Mode.

Third-Party. These functions allow data to be associated with a particular authentication request, yet not to be processed by the RSA ACE/Server.

Housekeeping. These functions initialize and later clean up system resources needed by the API. One set of housekeeping functions is called only once and is used for internal data structures, worker threads, and to load into memory the **sdconf.rec** and **sdopts.rec** files (documented in the RSA ACE/Server documentation set). Another set of functions is used for each authentication request to initialize a socket, verify communication with an RSA ACE/Server, manage a request-specific data structure, and enable threads for further processing of a particular request.

Miscellaneous. The **AceGetShell** and **AceGetTime** functions are included in this category.

Figure 2 lists the relevant functions in each of these categories. This taxonomy is intended to help developers think through the type of processing that must be accomplished by an Agent, and to select the appropriate functions for that processing. This categorization does not reflect all the details of which functions can be used together, or the order in which they should be called. This taxonomy does not take the place of the detailed descriptions of the functions contained later in this guide, nor can it substitute for reviewing the sample source code.

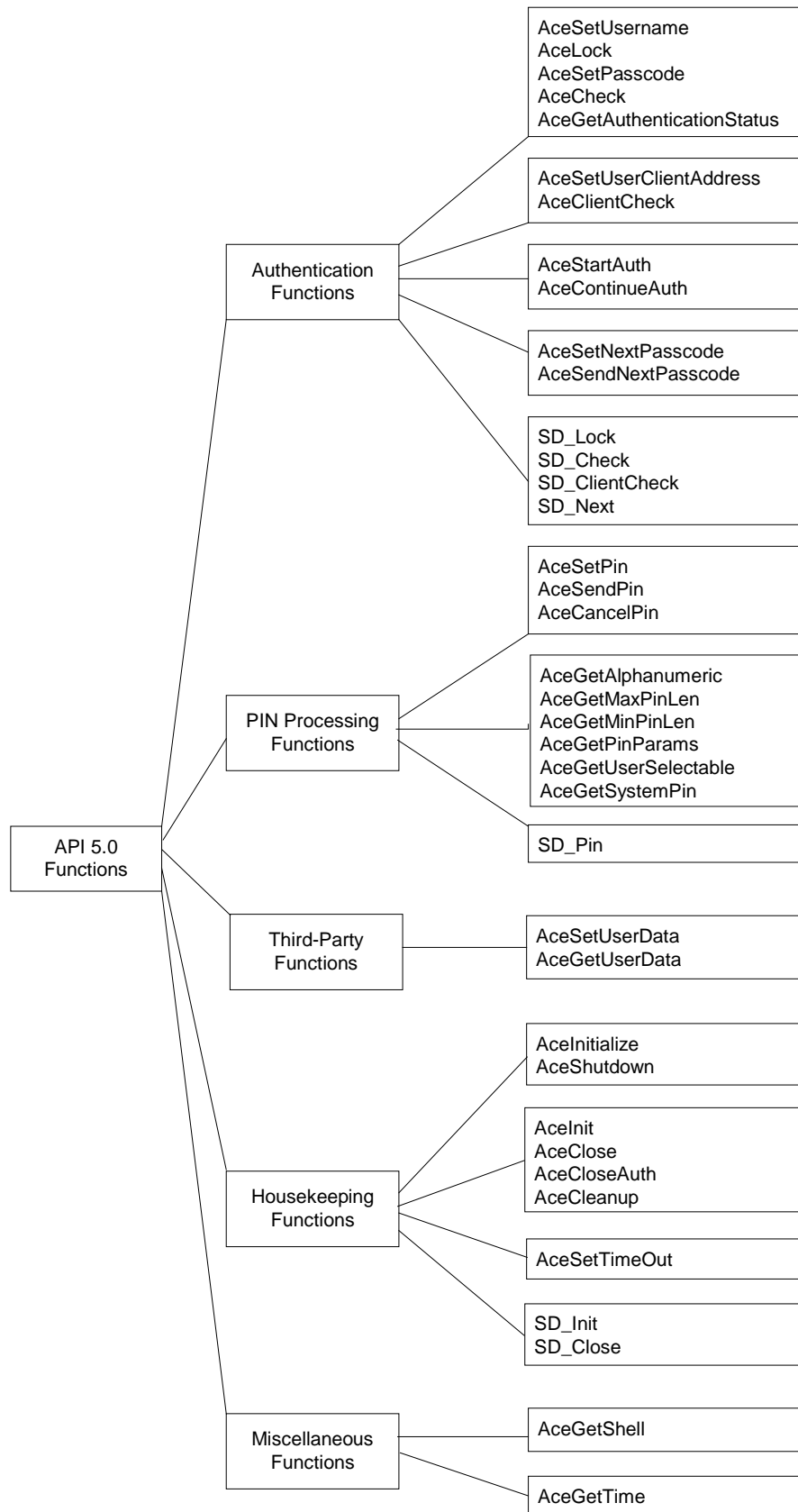


Figure 2: API 5.0 Functional Groups

Synchronous and Asynchronous Functions

The Authentication API includes both synchronous and asynchronous functions. All of the API functions are synchronous, except for the following functions, which are asynchronous:

- The Authentication functions **AceLock**, **AceCheck**, **AceClientCheck** and **AceSendNextPasscode**
- The PIN Processing functions **AceSendPin** and **AceCancelPin**
- The Housekeeping functions **AceInit** and **AceClose**.

Because asynchronous API functions can return before completing their work, you must supply your own callback functions to determine the status of the actions initiated by an asynchronous call. Callback functions are called when the asynchronous function finishes. You can find an asynchronous code sample file, named **async**, in the **\ACEAgentSDK\samples** directory on the distribution CD-ROM.

In general, those Agents that provide an authentication service to multiple users or clients should use the asynchronous functions, because such Agents should not block while waiting for a response from the RSA ACE/Server. Agents that service one requestor at a time, and that can therefore block while waiting for a response to a call on the RSA ACE/Server, will likely only need the synchronous functions provided by the Authentication API. As noted in “Appendix B: Using Synchronous and Asynchronous API Functions,” it is possible to mix some aspects of the asynchronous and synchronous calls.

Compiling and Linking with the RSA ACE/Agent Authentication API 5.0

Compiling on UNIX and Windows NT

To compile an Agent using version 5.0 of the Authentication API, developers must include the file **acexport.h** in their source, which defines the appropriate forms of the API entry points. If compiling under UNIX (Solaris, HP-UX, or AIX), the symbol **UNIX** must be defined with the **-DUNIX** compiler option. If compiling under Windows NT, the symbol **WIN32** must be defined with the **-DWIN32** compiler option.

The detailed descriptions contained in “API Functions” on page 23 show the Windows NT form of the API entry point prototypes, and therefore refer to the “WINAPI” calling convention for Windows NT. The **acexport.h** file contains the correct form of the prototypes based on the platform symbol definition, so the same descriptions are relevant for UNIX environments.

When linking objects with the API library, the correct platform version of the API library must be selected from the directory **ACEAgentSDK/lib**. Under Windows NT, the developer can choose to use the export library **aceclnt.lib** or compile the runtime linkage code, **aceauth.c**, provided under the **ACEAgentSDK/src** directory. To use this code it is necessary to include the file **aceauth.h** in place of the **acexport.h** file.

Using Dynamic Libraries or Static Libraries in UNIX

This release of the API library provides both dynamic and static (archive) libraries for all supported UNIX platforms (Solaris, HP-UX, or AIX). When using dynamic libraries to build Agent applications, it is important to take precautions to prevent users from substituting a different version of the **libaceclnt.so** library file in place of the one with which the code was compiled. A preferred means of accomplishing this is to use the full path to the library in the command line passed to the **ld** command. Doing so will ensure that a specific version of the library will be loaded, instead of one resulting from a search of the library path. If the version of the dynamic linker used provides options that disable the library path processing at runtime, it is advisable to use them as well. See the makefiles provided in **\ACEAgentSDK\samples** for examples of these techniques.

For information about downloading example files from the RSA Security Web site, see “Sample Source Code” on page 6.

API Functions

This section explains each function in the RSA ACE/Agent Authentication Application Programming Interface (API) in detail. As discussed in “Compiling on UNIX and Windows NT” on page 20, the entry point prototypes contain the “WINAPI” calling convention of Windows NT, but can be compiled for both Windows and UNIX. For additional information on using the APIs, refer to “Compiling and Linking with the RSA ACE/Agent Authentication API 5.0” on page 20.

Important: When reviewing the information in this section, it is important to distinguish between **return values** and **result codes**. The term *return value* has the traditional meaning common to languages like C and C++. The term *result code*, also referred to as a *status code*, refers to the setting of flags that can only be checked by using the **AceGetAuthenticationStatus** function. Many of the API functions provide a return value that indicates whether the function successfully completed. However, it is only by checking the result code with **AceGetAuthenticationStatus** that you can determine whether a user successfully authenticated when asynchronous function calls were used. A complete list of return values, result codes, and their meaning is contained in Appendix A.

The specifics of using the various functions described in this section are demonstrated in sample programs that are provided on the CD-ROM, and on **<http://securcare.rsasecurity.com>**. Developers of Agents should closely examine the source code for each of the samples to understand the proper calling sequences required. Each sample demonstrates a different set of functions. Additional information about the sample programs can be found in “Sample Source Code” on page 6.

AceCancelPin

Description

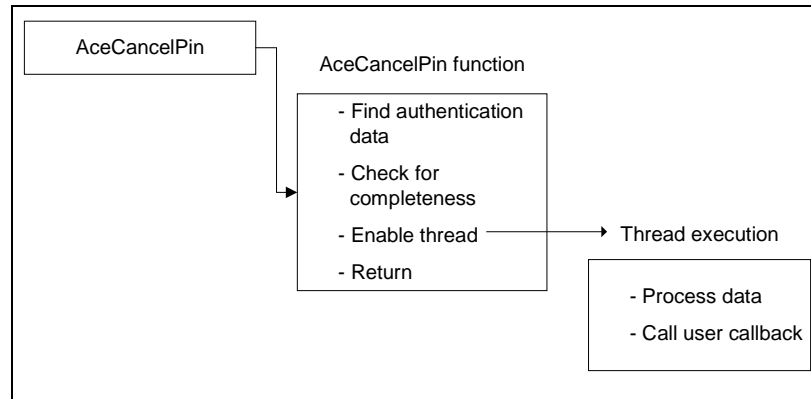
```
int WINAPI AceCancelPin(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE SdiHandle));
```

The **AceCancelPin** function cancels the pending New PIN operation associated with SdiHandle. **AceCancelPin** should be called only when the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED** and the user has chosen not to proceed through New PIN Mode.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, threads are enabled to process the data.

The following figure illustrates what happens when **AceCancelPin** is called.



The **AceCancelPin** function attempts to find the data associated with this specific authentication process through the unique handle value. If the data is found, **AceCancelPin** calls another thread to process the data. When the threads invoked by **AceCancelPin** finish, the developer-defined function is called (if one has been supplied). **AceCancelPin** allows caller to defer changing the PIN in the RSA ACE/Server token record.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceCancelPin could not locate the data associated with the handle.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will have the following value:

ACM_OK	The PIN operation was successfully aborted.
---------------	---

AceCheck

Description

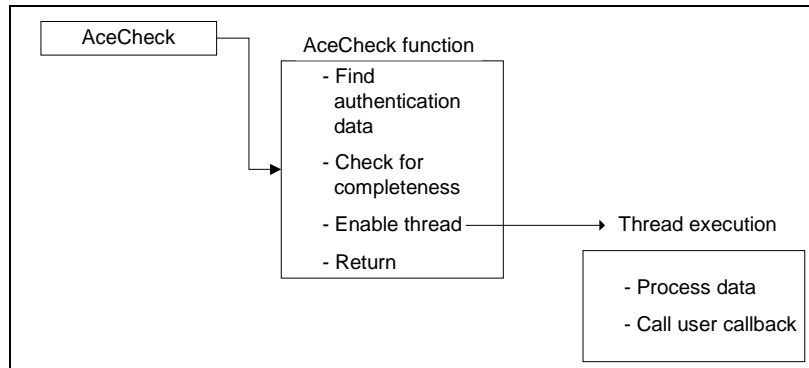
```
int WINAPI AceCheck(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE));
```

The **AceCheck** function checks the validity of the **passcode** (previously set by a call to **AceSetPasscode**) for the **username** (previously set by a call to **AceSetUsername**). **AceCheck** should be called only after successful calls to **AceInit**, **AceSetPasscode**, and **AceSetUsername** for the specific authentication request.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, other threads perform the rest of the work.

The following figure illustrates what happens when you makes a call to **AceCheck**.



The **AceCheck** function attempts to find the data associated with this specific authentication process through the unique handle value. If the data is found, **AceCheck** enables other threads to process the data. When the threads invoked by **AceCheck** finish, the developer-defined function is called (if one has been supplied).

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceCheck could not locate the data associated with the handle.
ACE_UNDEFINED_PASSCODE	The PASSCODE has yet to be set for the authentication request.
ACE_UNDEFINED_USERNAME	The username has yet to be set for the authentication request.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_OK	The user successfully authenticated. Use the AceGetShell function to access the shell field.
ACM_ACCESS_DENIED	The user failed authentication.
ACM_NEXT_CODE_REQUIRED	Next tokencode required. Use AceSetNextPasscode and AceSendNextPasscode to complete the transaction.
ACM_NEW_PIN_REQUIRED	New PIN required. You can use the AceGetxxx functions to access the PIN limits. Use AceSendPin or AceCancelPin to complete the transaction.

AceCleanup

Description

```
void WINAPI AceCleanup(  
    void (WINAPI*appCallback) (SDI_HANDLE) );
```

The **AceCleanup** function can be called at any time to discard every authentication request in progress. After this call has returned, every handle that has been returned from a call to **AceInit**, **AceStartAuth**, or **SD_Init** will be invalid. The callback argument will be used to clean up before the authentication handle is discarded.

If the callback passed as an argument to **AceCleanup** is null, then any callback previously defined by a call to **AceSetTimeout** will be invoked.

Architecture

This synchronous function traverses the internal list of outstanding authentication handles and performs the cleanup operation on them. This has the same effect as if the user's actions had caused the appropriate final call (**AceClose**, **AceCloseAuth**, or **SD_Close**) to be called.

Input Argument

appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.
--------------------	---

Outputs and Post Conditions

The function does not return a value. When the function call returns, every outstanding authentication has been discarded, and all handles that have been returned by **AceInit**, **AceStartAuth**, or **SD_Init** will be invalid.

AceClientCheck

Description

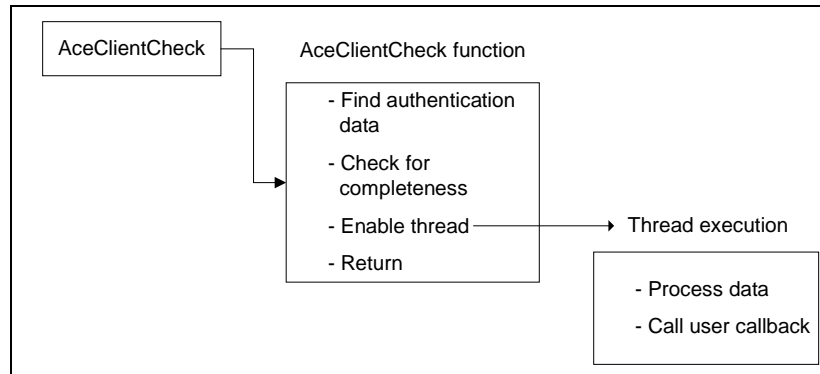
```
int WINAPI AceClientCheck(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE));
```

The **AceClientCheck** function checks the validity of the PASSCODE (previously set by a call to **AceSetPasscode**) for the **username** (previously set by a call to **AceSetUsername**) on behalf of another client. The other client's IP address must have been set with a call to **AceSetUserClientAddress**. **AceClientCheck** should be called only after successful calls to **AceInit**, **AceSetPasscode**, **AceSetUsername**, and **AceSetUserClientAddress** for the same authentication request.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, other threads perform the rest of the work.

The following figure illustrates what happens when a call is made to **AceClientCheck**.



The **AceClientCheck** function attempts to find the data associated with the specific authentication process through the unique handle value. If the data is found, **AceClientCheck** checks to make sure all the data it needs to attempt authentication is present, then enables another thread to process the data. When the threads invoked by **AceClientCheck** finish, the developer-defined function is called (if one has been supplied).

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceCheck could not locate the data associated with the handle.
ACE_UNDEFINED_PASSCODE	The PASSCODE has yet to be set for the authentication request.
ACE_UNDEFINED_USERNAME	The username has yet to be set for the authentication request.
ACE_UNDEFINED_CLIENTADDR	The client address has yet to be set for the authentication request.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_OK	The user successfully authenticated. Use the AceGetShell function to access the shell field.
ACM_ACCESS_DENIED	The user failed authentication.
ACM_NEXT_CODE_REQUIRED	Next tokencode required. Use AceSetNextPasscode and AceSendNextPasscode to complete the transaction.
ACM_NEW_PIN_REQUIRED	New PIN required. You can use the AceGet function to access the PIN limits. Use AceSendPin or AceCancelPin to complete the transaction.

AceClose

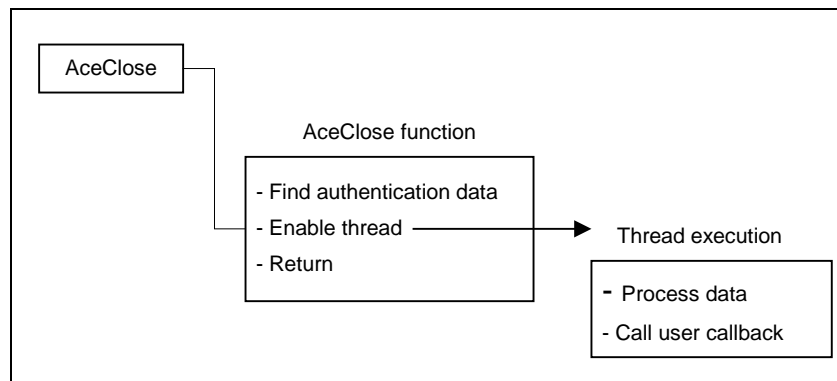
Description

```
int WINAPI AceClose(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback)(SDI_HANDLE SdiHandle));
```

The **AceClose** function closes the socket and frees the data associated with the authentication process specified by **SdiHandle**. This function should be called after attempting to authenticate the user regardless of whether or not the authentication was successful. Note that if **ACE_PROCESSING** was never returned from a call to **AceInit**, the **AceClose** function need not be called for that particular authentication request.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, threads are enabled to finish processing the data.



The **AceClose** function attempts to find the data associated with this specific authentication process through the unique handle value. If the data is found, **AceClose** calls another thread of execution to process the data. When the threads invoked by **AceClose** finish, the developer-defined function is called (if one has been supplied).

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceClose could not locate the data associated with the handle.

Note: The handle used in the **AceClose** call cannot be accessed after the callback has been invoked. Since **AceGetUserData** requires a valid handle, an attempt to access **UserData** for a specific authentication request will fail after **AceClose** has freed the resources associated with that request. However, the callback function supplied to **AceClose** will be able to access the data. If you must dispose of the data, either do so before calling **AceClose** or at the callback.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be the following value:

ACM_OK	The authentication handle was successfully disposed of.
---------------	---

AceCloseAuth

Description

```
SD_ERROR WINAPI AceCloseAuth(  
    SDI_HANDLE SdiHandle)
```

The **AceCloseAuth** function closes down an authentication context and frees up any memory that was allocated on its behalf. It can be called any time after a successful return from **AceStartAuth** for the particular authentication context specified by the **SDI_HANDLE** value. This function is the final step in the synchronous API model.

Note: If you use the version 5.0 of the RSA ACE/Agent Authentication API with existing Agents, you do not have to change your use of the **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth** functions to support two-step authentication. Two-step authentication is automatically performed within these three functions.

Architecture

The caller of this synchronous function must supply as the first argument an **SDI_HANDLE** value that was previously set in the call to **AceStartAuth**.

Input Argument

SdiHandle	An SDI_HANDLE associated with the authentication context.
------------------	--

Outputs and Post Conditions

If the return value is **ACM_OK**, it is successful. Otherwise, it is any of the error values returned by the **AceClose** asynchronous API call.

Note: The handle used in the **AceCloseAuth** call will be invalid after the function returns. To retrieve the result of the authentication, call **AceGetAuthenticationStatus** *before* calling **AceCloseAuth**. If **AceGetAuthenticationStatus** is mistakenly called immediately after **AceCloseAuth**, it will return the status of the close, not of the authentication.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

AceContinueAuth

Description

```
SD_ERROR WINAPI AceContinueAuth(
    SD_HANDLE SdiHandle,
    char *resp,
    SD_I32 respLen,
    SD_BOOL *moreData,
    SD_BOOL *echoFlag,
    SD_I32 *respTimeout,
    SD_I32 *nextRespLen,
    char *promptStr,
    SD_I32 *promptStrLen)
```

The **AceContinueAuth** function supplies the next value needed by the authentication context. It is the middle step in the synchronous API model that begins with **AceStartAuth** and ends with **AceCloseAuth**.

The values of the prompts returned by this function are stored in a message catalog that can be modified by an Agent developer. See “Appendix C: Modifying the Message Catalog” on page 79 for additional information.

Note: If you use the version 5.0 of the RSA ACE/Agent Authentication API with existing Agents, you do not have to change your use of the **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth** functions to support two-step authentication. Two-step authentication is automatically performed within these three functions.

Architecture

Your code must continue to supply data to the authentication context through this function for as long as this function returns with the **moreData** argument set to **True**. The **promptStr** argument will contain the string that should be used to display the next response message to the user who is attempting authentication. This string, which is set by the API itself, will contain either a prompt for the user (for example, “Enter next tokencode:”) or a statement intended to provide status information to the user (for example, “Access denied.”).

Note: You must use the **AceGetAuthenticationStatus** function with the handle returned from **AceStartAuth** to get the actual authentication status. The return value of this function simply indicates the success or failure of the function, not of the authentication in progress.

Input Arguments

SdiHandle	The SD_HANDLE whose value is filled in by the AceStartAuth function.
resp	A pointer to the string containing the response value.
respLen	An integer containing the length of the response string.
moreData	A flag that indicates whether or not more data is needed by the authentication context.
echoFlag	A flag that gives a hint to the developer as to whether or not the next expected response should be echoed to the screen.

respTimeout	A hint to the developer about how long to display the next prompt string.
nextRespLen	Indicates the maximum number of bytes of data expected in the next developer-supplied response.
promptStr	Developer-supplied character array to be filled in with the string to be used as the next message displayed to the user.
promptStrLen	Initially set to the size of the developer-supplied storage for the promptStr , its value becomes the length of the filled-in promptStr string.

Outputs and Post Conditions

The return value indicates whether or not the function was successful in copying the data pointed to by the response. An **ACM_OK** value is returned on success, and an error value is returned upon failure. Typical return values are:

ACM_OK	The call completed successfully. Use AceGetAuthenticationStatus to determine the success or failure of the authentication.
ACE_INVALID_ARG	Could not set the data pointed to by response as the next expected data value.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACE_NOT_ENOUGH_STORAGE	The size of the developer-supplied promptStr is too small.

Error Handling

Your code must close down the authentication context using **AceCloseAuth** regardless of whether this function succeeds.

Checking the Results

Calling the **AceGetAuthenticationStatus** function allows the caller to verify the actual status of the authentication operations. The status codes are defined in the description of the **AceGetAuthenticationStatus** function.

AceGetAlphanumeric

Description

```
int WINAPI AceGetAlphanumeric(  
    SDI_HANDLE SdiHandle,  
    char *val)
```

The **AceGetAlphanumeric** function determines whether the PIN of the authenticating user can have alphanumeric characters. It is helpful to retrieve this value when the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED** and the Agent is designed to validate that the new PIN entered by the user conforms to the required PIN characteristics set by the RSA ACE/Server administrator. Any such checking done by the Agent is simply as a convenience to the user. The PIN will be fully checked for correctness by the RSA ACE/Server.

Architecture

This synchronous function fills in the second argument passed to it with a value that indicates whether or not a user's new PIN can have alphanumeric characters.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character that will be assigned the alphanumeric flag value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the alphanumeric value was filled in by the function. This value can be one of the following:

0	The user's PIN can have only numeric values.
1	The user's PIN can have alphabetic or numeric values.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetAuthenticationStatus

Description

```
int WINAPI AceGetAuthenticationStatus(
    SDI_HANDLE SdiHandle,
    INT32BIT *val)
```

The **AceGetAuthenticationStatus** function determines the status of the authentication request at each step in the process. This function is very useful as part of a callback function because an Agent cannot directly access the **SD_CLIENT** data.

Architecture

This synchronous function fills in the current value for the result code associated with the authentication request data.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to data that will hold the result code value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the result code was filled in by the function. A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

Some of the more common values and meanings for the result code are:

ACM_OK	The meaning of ACM_OK depends on the function that caused it to be set as a result code. See the table in “Appendix A: Return Values and Result Codes” on page 73 for details.
ACM_NEXT_CODE_REQUIRED	Another PASSCODE must be supplied before authentication can be granted.
ACM_ACCESS_DENIED	The user’s authentication failed.
ACM_NEW_PIN_REQUIRED	The token is in New PIN Mode.
ACM_NEW_PIN_ACCEPTED	The RSA ACE/Server has accepted a new PIN. The user should now be required to authenticate with the new PIN.
ACM_NEW_PIN_REJECTED	The RSA ACE/Server rejected the new PIN. The PIN might not have matched the parameters set in the return from AceGetPinParams .
ACM_NEXT_CODE_BAD	The PASSCODE supplied to the RSA ACE/Server is not valid.

Important: There is a design constraint on the use of **SDI_HANDLE**. An asynchronous application of the API *must* avoid concurrent use of the *same* **SDI_HANDLE**. Multiple threads using the same **SDI_HANDLE** will return incorrect results.

Error Handling

To handle errors appropriately, use the values set by this function at decisions points in your code.

AceGetMaxPinLen

Description

```
int WINAPI AceGetMaxPinLen(  
    SDI_HANDLE SdiHandle,  
    char *val)
```

The **AceGetMaxPinLen** function determines the value of the maximum PIN length allowed by the RSA ACE/Server. It is helpful to retrieve this value when the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED** and the Agent is designed to validate that the new PIN entered by the user conforms to the required PIN characteristics set by the RSA ACE/Server administrator. Any such checking done by the Agent is simply as a convenience to the user. The PIN will be fully checked for correctness by the RSA ACE/Server.

Architecture

This function is synchronous. It uses the second argument passed to it to fill in the current value for the maximum PIN length associated with the authentication request data.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character that will be assigned the maximum PIN length value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the maximum PIN length currently allowed by the RSA ACE/Server was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetMinPinLen

Description

```
int WINAPI AceGetMinPinLen(  
    SDI_HANDLE SdiHandle,  
    char *val)
```

The **AceGetMinPinLen** function determines the value of the minimum PIN length allowed by the RSA ACE/Server. It is helpful to retrieve this value when the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED** and the Agent is designed to validate that the new PIN entered by the user conforms to the required PIN characteristics set by the RSA ACE/Server administrator. Any such checking done by the Agent is simply as a convenience to the user. The PIN will be fully checked for correctness by the RSA ACE/Server.

Architecture

This synchronous function uses the second argument passed to it to fill in the current value for the minimum PIN length associated with the authentication request data.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character that will be assigned the minimum PIN length value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the minimum PIN length currently allowed by the RSA ACE/Server was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetPinParams

Description

```
int WINAPI AceGetPinParams (
    SDI_HANDLE SdiHandle,
    SD_PIN *val)

typedef struct
{
    SD_CHAR Min;
    SD_CHAR Max;
    SD_CHAR Selectable;
    SD_CHAR Alphanumeric;
    SD_CHAR System[LENPRNST+2];
} SD_PIN;
```

The **AceGetPinParams** function obtains all of the PIN-related parameters at once. It is helpful to retrieve this value when the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED** and the Agent is designed to validate that the new PIN entered by the user conforms to the required PIN characteristics set by the RSA ACE/Server administrator. Any such checking done by the Agent is simply as a convenience to the user. The PIN will be fully checked for correctness by the RSA ACE/Server.

Architecture

The caller of this synchronous function must supply, as the second argument, a pointer to a structure of type **SD_PIN**, into which the function will copy the PIN parameters. **AceGetPinParams** does not allocate storage for the data on its own.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
Val	A pointer to a structure of type SD_PIN that will contain copies of the PIN parameters.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the PIN value was provided by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetShell

Description

```
int WINAPI AceGetShell(  
    SDI_HANDLE SdiHandle,  
    char *shell)
```

The **AceGetShell** function gets a copy of the string value containing the user's shell. This value can be retrieved at any time after a successful call to **AceCheck** or **AceClientCheck** because that is when the value is retrieved from the RSA ACE/Server.

Architecture

This function is synchronous and the caller must supply, as the second argument, a pointer to a character array into which the value will be copied.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
shell	A pointer to developer-supplied storage that will receive a copy of the user data.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the data value was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetSystemPin

Description

```
int WINAPI AceGetSystemPin(  
    SDI_HANDLE SdiHandle,  
    char *val)
```

The **AceGetSystemPin** function gets a copy of a PIN generated by the RSA ACE/Server for the authentication request associated with **SdiHandle**. The retrieval of this value might be necessary if the result code from **AceCheck** is **ACM_NEW_PIN_REQUIRED**, and a call to **AceGetPinParams** or **AceGetUserSelectable** indicates that a system-generated PIN has been required by the RSA ACE/Server Administrator.

Architecture

The caller of this synchronous function must supply, as the second argument, a pointer to a character string into which the PIN string will be copied. This function does not allocate storage for this data on its own.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character string that will receive a copy of the PIN.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the PIN value was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetTime

Description

```
int WINAPI AceGetTime(  
    SDI_HANDLE SdiHandle,  
    UINT32BIT *val)
```

The **AceGetTime** function gets a copy of the current time plus the time delta between the Agent and the RSA ACE/Server. The retrieval of this value can be performed at any time after a successful call to **AceInit**. The value is reported in seconds.

Architecture

This function is synchronous and the caller must supply, as the second argument, a pointer to a 4-byte storage area into which the time value will be copied.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to developer-supplied storage that will receive a copy of the user data.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the data value was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetUserData

Description

```
int WINAPI AceGetUserData(
    SDI_HANDLE SdiHandle,
    unsigned int *val)
```

The **AceGetUserData** function gets a copy of the user data previously set as the value of the second argument in the call to **AceInit** or **AceSetUserData**. This value can be retrieved at any time. The RSA ACE/Server will not process data stored with this function.

The purpose of the calls **AceSetUserData** and **AceGetUserData** is to supply Agent developers with a convenient location and associated routines for storing and retrieving data related to the authentication in progress. Although this is most useful in cases of true asynchronous operation, it is available to both the synchronous calls and the asynchronous calls.

An example of a use for these calls is in an application that operates within a server. The application might have to maintain information about the user authentication. In that case, the code should allocate some memory and pass a pointer to the memory as the 32-bit data reference to **AceSetUserData**. When this data is needed at a later point in the authentication process, the code need only maintain the **SDI_HANDLE** value. To get access to the data, the code would then call **AceGetUserData** to retrieve the pointer to the original data and process it. The caller of these functions is responsible for the disposition of any resources associated with the data referenced by the 32-bit value. The Authentication API will furnish the value any time it is requested.

Architecture

This function is synchronous and the caller must supply, as the second argument, a pointer to a 32-bit storage area (that is, an **unsigned int**) into which to copy the user data value.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a 32-bit storage area (an unsigned int) that will receive a copy of the user data.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the data value was filled in by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceGetUserSelectable

Description

```
int WINAPI AceGetUserSelectable (
    SDI_HANDLE SdiHandle,
    char *val)
```

The **AceGetUserSelectable** function determines whether or not the user associated with the authentication request has the ability to select a new PIN value. The retrieval of this value might be necessary when a value of **ACM_NEW_PIN_REQUIRED** is returned by **AceCheck**.

Architecture

This synchronous function fills in the second argument passed to it with a value that indicates whether or not the user can select a new PIN.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character that will be assigned the user-selectable value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the user-selectable value was filled in by the function. The value can be one of the following values:

CANNOT_CHOOSE_PIN	The user must either accept a system-generated PIN or cancel the operation and leave the token in New PIN Mode.
MUST_CHOOSE_PIN	The user must create his or her own PIN. The user is not given the option of receiving a system-generated PIN.
USER_SELECTABLE	The user can either create a PIN or request a system-generated one.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceInit

Description

```
int WINAPI AceInit(
    LP_SDI_HANDLE pSdiHandle,
    unsigned int userData,
    void (WINAPI*appCallback) (SDI_HANDLE));
```

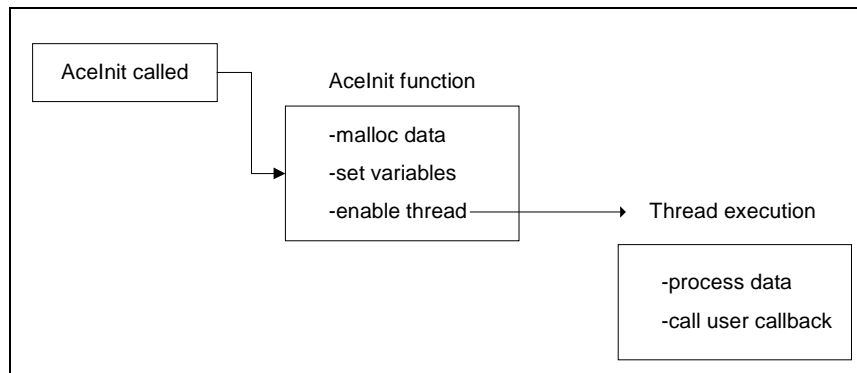
AceInit initializes a socket and makes a call to the RSA ACE/Server to verify communication. **AceInit** must be called first in the sequence of calls for each authentication request.

The value of the argument **userData** can be retrieved with **AceGetUserData**. See the description of that function on page 44 for a discussion of the uses of **userData**.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, other threads complete the initialization process.

The following figure illustrates what happens when a call is made to **AceInit**.



The **AceInit** function allocates storage for a data structure that is to be used for a particular authentication process, sets variables within that structure, enables other threads, and returns a value to the caller. When the threads invoked by **AceInit** finish, the developer-defined function is called (if one has been supplied).

The developer passes a pointer to a handle and a function pointer to **AceInit**. **AceInit** assigns a unique value to the handle. In subsequent API calls, this handle identifies the particular authentication request. The developer can supply a callback function with the following prototype:

```
void WINAPI Callback(SDI_HANDLE);
```

The callback function is called when **AceInit** finishes.

Input Arguments

pSdiHandle	A pointer to a handle that will be assigned by the AceInit function.
userData	A 32-bit value that will hold data to be associated with the authentication request. It can be passed a handle or any other pointer type.
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed to AceInit instead of a valid function pointer.

Outputs and Post Conditions

The value returned by the **AceInit** function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_INIT_NO_RESOURCE	Attempt to allocate memory failed.
ACE_INIT_SOCKET_FAIL	Attempt to create socket failed.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue. If **ACE_PROCESSING** is returned, your code must eventually call **AceClose**.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_OK	The connection with the RSA ACE/Server was successful. Use AceCheck or AceClientCheck to proceed with the authentication.
ACM_NO_SERVER	The connection with the RSA ACE/Server failed. End the authentication attempt with AceClose .

AceInitialize

Description

```
SD_BOOL WINAPI AceInitialize( void );
```

AceInitialize initializes the worker threads and loads into memory the mandatory configuration file **sdconf.rec** that was created during installation of the RSA ACE/Server software, and the optional configuration file **sdopts.rec**. In the Windows NT environment, the API expects to find the configuration files in the *%SystemRoot%\system32* directory. In the UNIX environment, the API expects to find the configuration files in the */var/ace* directory (or the directory specified by the **\$VAR_ACE** system variable). Make certain that the configuration files are present in the directory that corresponds to your environment.

AceInitialize must be called before any other API function is called, but it should be called only once. The **AceInitialize** function has replaced the decremented, no longer supported **creadcfg** call. RSA Security *strongly* recommends that you replace any **creadcfg** calls with calls to **AceInitialize**.

Important: In earlier versions of the Authentication API, calls to **AceInitialize** were necessary only for the UNIX library to initialize the threads on UNIX and to read data from the RSA ACE/Server configuration file, **sdconf.rec**. Now your code must also call **AceInitialize** in the Windows NT environment. RSA Security *requires* that your code always call **AceInitialize** once, before calling any other API function.

Although it is the developer's responsibility to call **AceInitialize**, to developers avoid potential error conditions, **AceInit**, **AceStartAuth**, **SD_Init**, and **sd_init** will detect whether **AceInitialize** has already been called, and will call it if necessary. However, if your application calls **AceInit**, **AceStartAuth**, or **SD_Init** in a multithreaded environment, and multiple automatic calls to **AceInitialize** happen at the same time, those functions will return an error. In that case, your code must make a call to **AceInitialize** once *before* calling **AceInit**, **AceStartAuth**, or **SD_Init**.

Architecture

This function will create the internal structures and the worker threads that will operate during the authentication requests. If it is called more than once, it will simply return success. If more than two threads call this function for the first time at the same time, one of the calls will return **SD_FALSE**. This can happen if your code does not call **AceInitialize** directly and instead relies on the automatic call from **AceInit**, **AceStartAuth**, **SD_Init**, or **sd_init**. (See the Important note under "Description" for more information.)

Input Arguments

None.

Outputs and Post Conditions

If this function initializes the internal state of the library successfully, it will return **SD_TRUE**. If there is any failure, it will return **SD_FALSE**.

AceLock

Description

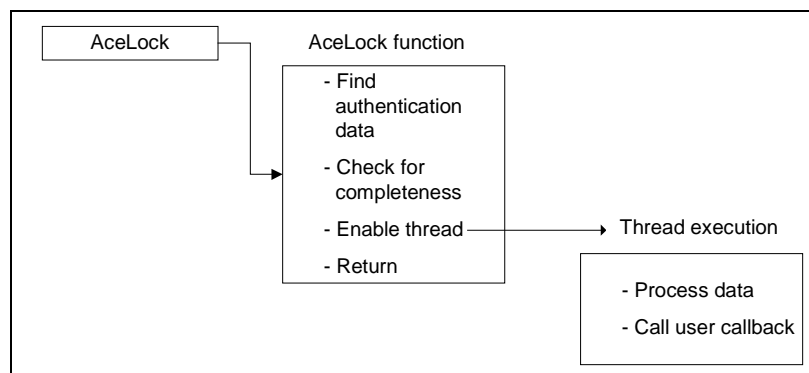
```
int WINAPI AceLock(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE));
```

The **AceLock** function verifies that a non-empty **username** has been set by a previous call to **AceSetUsername**, and then sends a lock request to the RSA/ACE Server as part of the two step authentication process (see “New Two-Step Authentication Support” on page 7 for additional information). **AceLock** should be called only after successful calls to **AceInit** and **AceSetUsername** for the specific authentication request.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, other threads perform the rest of the work.

The following figure illustrates what happens when your code makes a call to **AceLock**.



The **AceLock** function attempts to find the data associated with this specific authentication process through the unique handle value. If the data is found, **AceLock** enables other threads to process the data. When the threads invoked by **AceLock** finish, the developer-defined function is called (if one has been supplied).

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceLock could not locate the data associated with the handle.
ACE_UNDEFINED_USERNAME	The username has yet to be set for the authentication request.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_OK	<p>The username lock request has been sent to the RSA ACE/Server.</p> <p>Use the AceCheck function to process the PASSCODE.</p>
ACM_ACCESS_DENIED	<p>Communication with the RSA ACE/Server has failed.</p> <p>The failure might be the result of a communication timeout or because the node secret between the Agent and the Server is mismatched. Work with the RSA ACE/Server administrator to troubleshoot the problem.</p>

AceSendNextPasscode

Description

```
int WINAPI AceSendNextPasscode (
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE));
```

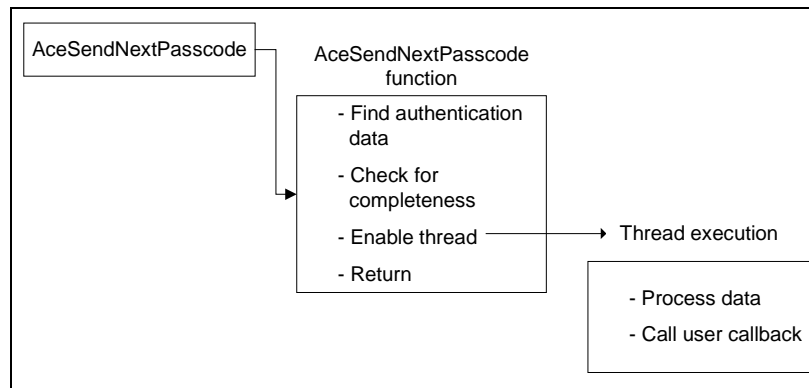
AceSendNextPasscode takes a successive tokencode and checks its validity. The next **passcode** value must have been previously set by a call to **AceSetNextPasscode** before calling this function. The integrating application must perform all I/O because **AceSendNextPasscode** does not display the **Next Code** prompt.

This function should be called if the result code from **AceCheck** is **ACM_NEXT_CODE_REQUIRED**. Note that an **ACM_NEXT_CODE_REQUIRED** result code can only occur after **AceCheck** has finished. Therefore, the developer-defined callback can check for this return value from **AceCheck** using **AceGetAuthenticationStatus**. The immediate return value from **AceCheck** is never set to this value.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, other threads perform the rest of the work.

The following figure illustrates what happens when a call is made to **AceSendNextPasscode**.



The **AceSendNextPasscode** function attempts to find the data associated with the specific authentication process through the unique handle value. If the data is found, **AceSendNextPasscode** enables another thread to process the data. When the threads invoked by **AceSendNextPasscode** finish, the developer-defined function is called (if one has been supplied).

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceSendNextPasscode could not locate the data associated with the handle.
ACE_UNDEFINED_NEXT_PASSCODE	The next PASSCODE has not yet been set for the authentication request.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_OK	The user successfully authenticated. Use the AceGetShell function to access the shell field.
ACM_ACCESS_DENIED	The user failed authentication.

AceSendPin

Description

```
int WINAPI AceSendPin(
    SDI_HANDLE SdiHandle,
    void (WINAPI*appCallback) (SDI_HANDLE SdiHandle));
```

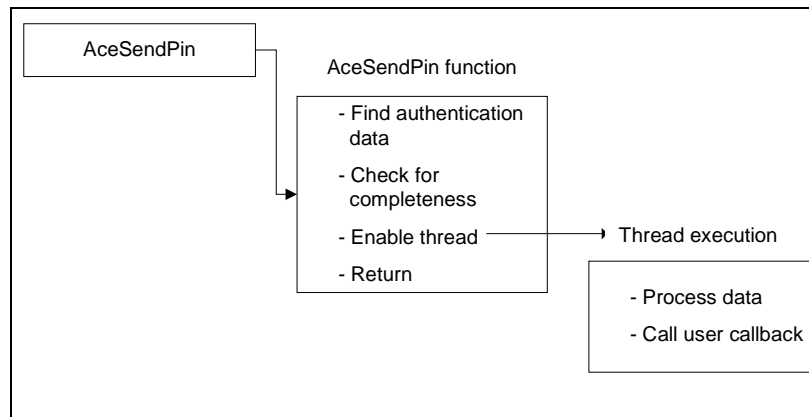
The **AceSendPin** function transmits a new PIN to the RSA ACE/Server for storage in a token record. **AceSendPin** should be called only when the result code **ACM_NEW_PIN_REQUIRED** is set by **AceCheck**.

Note: Do not consider users to be authenticated as soon as they have completed the New PIN operation. You must require that they authenticate using the new PIN.

Architecture

This asynchronous function returns immediately with a status value. If the status is **ACE_PROCESSING**, threads are enabled to process the data.

The following figure illustrates what happens when a call is made to **AceSendPin**.



The **AceSendPin** function attempts to find the data associated with the specific authentication process through the unique handle value. If the data is found, **AceSendPin** calls another thread to process the data. When the threads invoked by **AceSendPin** finish, the developer-defined function is called (if one has been supplied). **AceSendPin** allows the caller to attempt to change the PIN in the RSA ACE/Server token record. The PIN value must have been previously set with a call to **AceSetPin** before calling this function.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The value returned by this function will be one of the following values:

ACE_PROCESSING	Normal return.
ACE_ERR_INVALID_HANDLE	AceSendPin could not locate the data associated with the handle.
ACE_UNDEFINED_PIN	The user's PIN has not yet been set for the authentication request.

Error Handling

To handle errors appropriately, use the value returned by this function at a decision point in your code. A successful return should allow processing to continue.

Checking the Results

If the caller supplied a callback function, it will be able to verify the result of the authentication operation by calling the **AceGetAuthenticationStatus** function. The status codes will be one of the following values:

ACM_NEW_PIN_ACCEPTED	The RSA ACE/Server has accepted the new PIN. The user should now be required to authenticate with the new PIN.
ACM_NEW_PIN_REJECTED	The RSA ACE/Server rejected the new PIN. The PIN might not have matched the parameters set in the return from AceCheck .

AceSetNextPasscode

Description

```
int WINAPI AceSetNextPasscode(  
    SDI_HANDLE SdiHandle,  
    char *nextPasscode)
```

The **AceSetNextPasscode** function sets the **nextPasscode** value for the specific authentication request associated with the value of the handle passed to it. It should be called after a successful return from **AceCheck**. It should be used when the work completed by **AceCheck** revealed, using a call to **AceGetAuthenticationStatus**, that the Next Tokencode Mode is currently enabled. It must be called before calling **AceSendNextPasscode**.

Architecture

This function is synchronous.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
nextPasscode	A pointer to a character string containing the passcode value to be copied.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the next PASSCODE value was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found. A value of **ACE_INVALID_ARG** is returned if **nextPasscode** is NULL or its length is too small or too large.

AceSetPasscode

Description

```
int WINAPI AceSetPasscode(  
    SDI_HANDLE SdiHandle,  
    char *passcode)
```

The **AceSetPasscode** function sets the PASSCODE for the specific authentication request associated with the value of the handle passed to it. It should be called after a successful return from **AceInit** and before a call to **AceCheck**. **AceCheck** returns with an error if the PASSCODE value has not yet been set.

Architecture

The caller of this synchronous function must supply, as the second argument, a pointer to the character string containing the PASSCODE value to be copied.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
passcode	A pointer to a character string containing the PASSCODE value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and **passcode** was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found. A value of **ACE_INVALID_ARG** is returned if **passcode** is NULL or its length is too small or too large.

AceSetPin

Description

```
int WINAPI AceSetPin(  
    SDI_HANDLE SdiHandle,  
    char *PIN)
```

The **AceSetPin** function sets the PIN value for the specific authentication request associated with the value of the handle passed to it. It should be called after a successful return from **AceInit** and **AceCheck**. It should be used when the work completed by **AceCheck** revealed, using a call to **AceGetAuthenticationStatus**, that New PIN Mode is currently enabled for the token. It must be called prior to calling **AceSendPin**.

Architecture

This function is synchronous.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
PIN	A pointer to a character string containing the new PIN value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the new PIN value was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found. A value of **ACE_INVALID_ARG** is returned if new **PIN** value is NULL or its length is too small or too large.

AceSetTimeout

Description

```
int WINAPI AceSetTimeout(
    SDI_HANDLE SdiHandle,
    time_t lifetime,
    void (WINAPI*appCallback) (SDI_HANDLE SdiHandle))
```

The **AceSetTimeout** function has a dual purpose. When called with a **lifetime** set to a value other than 0, the function causes the Agent to perform **AceClose** processing automatically. The time is counted from the time of the last call to any of the API functions. When an **appCallback** pointer is passed, the API calls the function once the timeout occurs for performing cleanup operations that must be completed before the handle is discarded.

A secondary purpose of this function is to set up an **AceCleanup** callback. When the **AceCleanup** function is called, the callback will be used to clean up before the authentication request is discarded. An example of a cleanup callback would be a function that releases any open handles and frees any memory that might have been associated with the authentication as “user data.”

The **AceSetTimeout** function should be called only after a successful return from **AceInit**, **AceStartAuth**, or **SD_Init**.

Note: Do not call **AceClose**, **AceCloseAuth**, or **SD_Close** within the callback function. They are called as part of the timeout handling.

Architecture

This function is synchronous. You can supply a callback, but it will not be called unless a timeout occurs or the **AceCleanup** function is called.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
lifetime	The time in seconds after which the authentication handle is going to expire. A value of 0 can be passed if the authentication request is not going to expire.
appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the lifetime and cleanup callback was successfully set by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceSetUserClientAddress

Description

```
int WINAPI AceSetUserClientAddress(  
    SDI_HANDLE SdiHandle,  
    unsigned char *val)
```

The **AceSetUserClientAddress** function sets the IP address that will be considered the origin of the authentication request processed by a subsequent call to **AceClientCheck**.

Accordingly, a call to this function must be made before calling **AceClientCheck**.

AceClientCheck performs an authentication check just as **AceCheck** does, except that this request is done on behalf of the client machine whose IP address is set by a call to the **AceSetUserClientAddress** function.

The **AceSetUserClientAddress** function should be called only after a successful return from **AceInit**.

This function is useful in situations involving proxies.

Architecture

This function is synchronous.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
val	A pointer to a character array containing the IP address value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the IP address value was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceSetUserData

Description

```
int WINAPI AceSetUserData(
    SDI_HANDLE SdiHandle,
    unsigned int userData)
```

The **AceSetUserData** function sets the **userData** argument for the specific authentication request associated with the value of the handle passed to it. It should be called after a successful return from **AceInit**, **AceStartAuth**, or **SD_Init**. The data set by this function can be any 32-bit quantity, such as a pointer. It can be used to override the user data set by the **AceInit** call, or to supply similar data for the **AceStartAuth** or **SD_Init** calls. The data set by this call can be retrieved later by a call to **AceGetUserData**.

The purpose of the calls **AceSetUserData** and **AceGetUserData** is to supply Agent developers with a convenient location and associated routines for storing and retrieving data related to the authentication in progress. Although this is most useful in cases of true asynchronous operation, it is available to both the synchronous calls and the asynchronous calls.

An example of a use for these calls is in an application that operates within a server. The application might have to maintain information about the user authentication. In that case, the code would allocate some memory and pass a pointer to the memory as the 32-bit data reference to **AceSetUserData**. When this data is needed at a later point in the authentication process, the code need only maintain the **SDI_HANDLE** value. To access the data, the code would then call **AceGetUserData** to retrieve the pointer to the original data and process it. The caller of these functions is responsible for the disposition of any resources associated with the data referenced by the 32-bit value. The Authentication API will furnish the value any time it is requested.

Architecture

The caller of this synchronous function must supply, as the second argument, any 32-bit value to be copied.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
userData	A 32-bit value that will hold data to be associated with the authentication request. It can be passed a handle or any other pointer type.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the value of **userData** was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found.

AceSetUsername

Description

```
int WINAPI AceSetUsername(  
    SDI_HANDLE SdiHandle,  
    char *username)
```

The **AceSetUsername** function sets the **username** argument for the specific authentication request associated with the value of the handle passed to it. It should be called after a successful return from **AceInit** but before a call to **AceCheck**. **AceCheck** returns with an error if this value has not yet been set.

Architecture

The caller of this synchronous function must supply, as the second argument, a pointer to the character string containing the **username** value to be copied.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to AceInit .
username	A pointer to a character string containing the username value.

Outputs and Post Conditions

The function returns **ACE_SUCCESS** if the handle to the authentication data was found and the **username** was successfully copied by the function.

Error Handling

A value of **ACE_ERR_INVALID_HANDLE** is returned if the handle to the authentication data cannot be found. A value of **ACE_INVALID_ARG** is returned if **username** is NULL or its length is too large.

AceShutdown

Description

```
SD_BOOL WINAPI AceShutdown(  
    void (WINAPI*appCallback) (SDI_HANDLE) );
```

The **AceShutdown** function can be called when the application has completed its function and is about to terminate. After **AceShutdown** has been called, your code must call **AceInitialize** again before making any calls to the library. All calls will return **ACE_ERR_NOT_INITIALIZED** after **AceShutdown** completes. After this call has returned, every handle that has been returned from a call to **AceInit**, **AceStartAuth**, or **SD_Init** will be invalid.

The callback argument will be used to clean up before the authentication handle is discarded. If a callback has been defined by a call to **AceSetTimeout**, it will not be called if the callback for **AceShutdown** is not NULL. In that case, the callback passed as an argument to **AceShutdown** will be used instead.

Architecture

This synchronous function traverses the internal list of outstanding authentication handles and performs the cleanup operation on them. This has the same effect as if the user's actions resulted in the appropriate final call (**AceClose**, **AceCloseAuth**, or **SD_Close**) being called. In addition, both the internal worker threads and any memory or other resources will be released.

Input Arguments

appCallback	A pointer to a function written by the developer. This function returns nothing. The handle to the data area specific to the authentication process is passed to this function. A NULL value can be passed instead of a valid function pointer.
--------------------	---

Outputs and Post Conditions

The function returns **SD_TRUE** if it is successful and **SD_FALSE** if it fails. Upon return, every outstanding authentication has been discarded. All handles that have been returned by **AceInit**, **AceStartAuth**, or **SD_Init** will be invalid.

Error Handling

If the function is called more than once or before any calls to **AceInitialize**, it returns **SD_FALSE**.

AceStartAuth

Description

```
SD_ERROR WINAPI AceStartAuth(
    LP_SDI_HANDLE SdiHandle,
    char *userID,
    SD_I32 userIDLen,
    SD_BOOL *moreData,
    SD_BOOL *echoFlag,
    SD_I32 *respTimeout,
    SD_I32 *nextRespLen,
    char *promptStr,
    SD_I32 *promptStrLen)
```

The **AceStartAuth** function is designed to be used with **AceContinueAuth** and **AceCloseAuth**. These three functions provide a high-level model of abstracting the protocol and steps involved with authenticating users. The functions can be conceptually used with any number of challenge/response protocols.

The values of the prompts returned by this function are stored in a message catalog that can be modified by an Agent developer. See “Appendix C: Modifying the Message Catalog” on page 79 for more information.

Note: If you use the version 5.0 of the RSA ACE/Agent Authentication API with existing Agents, you do not have to change your use of the **AceStartAuth**, **AceContinueAuth**, and **AceCloseAuth** functions to support two-step authentication. Two-step authentication is automatically performed within these three functions.

Architecture

The **AceStartAuth** function provides the first step to take in trying to authenticate with our existing protocol using the synchronous API. If the function returns successfully, then your code calls **AceContinueAuth** with the required PASSCODE. Your code will continue to call **AceContinueAuth** until no more data is required by the authentication context, or an error occurs. Whenever **AceStartAuth** returns successfully, a call to **AceCloseAuth** must eventually be made for the specific authentication context.

Input Arguments

SdiHandle	A pointer to an SDI_HANDLE whose value will be specified by the function.
userID	A pointer to the string containing the username value.
userIDLen	An integer containing the length of the userID string.
moreData	A flag that will be set by the API to indicate whether or not more data is needed by the authentication context.
echoFlag	A flag that gives a hint to the developer as to whether the next expected response should be echoed to the screen.
respTimeout	A hint to the developer about how long to display the next prompt string for the user.

nextRespLen	Indicates the maximum number of bytes of data expected in the next developer-supplied response.
promptStr	Developer-supplied character array to be filled in by the API with the string that the caller should use as the next message displayed to the user.
promptStrLen	Initially set to the size of the developer-supplied storage for the promptStr , its value becomes the length of the filled-in promptStr string.

Outputs and Post Conditions

The function returns **ACM_OK** if successful. Otherwise, it returns with an error. Your code does not need to call **AceCloseAuth** if the function is not successful, but should call **AceCloseAuth** if the function is successful. Common error values are:

ACE_INIT_NO_RESOURCE	Memory allocation error.
ACE_EVENT_CREATE_FAIL	Could not create event object and associated data.
ACE_INIT_SOCKET_FAIL	Could not open and/or create a socket.
ACE_INVALID_ARG	Could not set the userID value.
ACE_NOT_ENOUGH_STORAGE	The size of the developer-supplied promptStr is too small.

Error Handling

The code does not have to call **AceCloseAuth** if this function does not succeed.

SD_Check

Description

```
int SD_Check(
    SDI_HANDLE SdiHandle,
    char *passcode,
    char *username)
```

SD_Check performs authentication by checking the validity of the PASSCODE entered by a user. **SD_Check** should only be called after successfully calling **SD_Init**. The integrating application must perform all I/O because **SD_Check** does not display the authentication prompts and messages.

This function is a synchronous wrapper for **AceCheck**, **AceSetUsername**, and **AceSetPasscode**.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
passcode	A pointer to the NULL terminated PASSCODE string.
username	A pointer to the NULL terminated username string. The username must contain fewer than 32 characters.

Return Values

Note: This function returns authentication result codes as return values.

ACM_OK	The user successfully authenticated. You can use the AceGetShell function to access the shell field.
ACE_UNDEFINED_PASSCODE	The passcode argument is an invalid length or is NULL.
ACE_UNDEFINED_USERNAME	The username argument is an invalid length or is NULL.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACM_ACCESS_DENIED	The user failed authentication.
ACM_NEXT_CODE_REQUIRED	Next tokencode required. Use SD_Next to complete the transaction.
ACM_NEW_PIN_REQUIRED	New PIN required. You can use the AceGetxxx functions to access the PIN limits. Use SD_Pin to complete the transaction.

SD_ClientCheck

Description

```
int SD_ClientCheck(
    SDI_HANDLE SdiHandle,
    char *passcode,
    char *username,
    unsigned long client_addr)
```

SD_ClientCheck performs authentication by checking the validity of the PASSCODE entered by a user. **SD_ClientCheck** should only be called after successfully calling **SD_Init**. The integrating application must perform all I/O because **SD_ClientCheck** does not display the authentication prompts and messages.

This synchronous function wraps the functionality provided by **AceSetUserName**, **AceSetPasscode**, **AceSetUserClientAddress**, and **AceClientCheck**.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
passcode	A pointer to the NULL-terminated PASSCODE string. The PASSCODE must contain 4 to 16 characters.
username	A pointer to the NULL-terminated username string. The username must contain fewer than 32 characters.
client_addr	A pointer to an IP address value.

Return Values

Note: This function returns authentication result codes as return values.

ACM_OK	The user successfully authenticated. You can use the AceGetShell function to determine the user's shell.
ACE_UNDEFINED_PASSCODE	The passcode argument is an invalid length or is NULL.
ACE_UNDEFINED_USERNAME	The username argument is an invalid length or is NULL.
ACE_UNDEFINED_CLIENTADDR	The client_addr argument is 0.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACM_ACCESS_DENIED	The user failed to authenticate successfully.
ACM_NEXT_CODE_REQUIRED	Next tokencode required. Use SD_Next to complete the transaction.
ACM_NEW_PIN_REQUIRED	New PIN required. You can use the AceGetxxx functions to get access to the PIN limits. Use SD_Pin to complete the transaction.

SD_Close

Description

```
int SD_Close(  
    SDI_HANDLE SdiHandle)
```

SD_Close releases any resources allocated by **SD_Init**. This function should be called after a successful call to **SD_Init**.

This function is a synchronous wrapper for **AceClose**.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
------------------	---

Return Values

Note: This function returns authentication result codes as return values.

ACM_OK	The authentication session has closed successfully.
ACE_ERR_INVALID_HANDLE	Handle value is invalid.

SD_Init

Description

```
int SD_Init(  
    LP_SDI_HANDLE pSdiHandle);
```

SD_Init initializes communication between the Agent and the RSA ACE/Server. It initializes the socket and makes a call to the RSA ACE/Server to verify communication. **SD_Init** must be called after a successful call to **AceInitialize**, but before calling any other API function.

This function is a synchronous wrapper for **AceInit**.

Input Arguments

pSdiHandle	A pointer to a handle that will be assigned by the SD_Init function.
-------------------	---

Return Values

ACM_OK	No error.
ACE_INIT_NO_RESOURCE	Attempt to allocate memory failed.
ACE_INIT_SOCKET_FAIL	Attempt to create socket failed.

Error Handling

If **SD_Init** returns **ACM_OK**, you must call **SD_Close** to release the resources associated with the authentication handle. If any error occurs, the handle will not get set, and you need not call **SD_Close**.

SD_Lock

Description

```
int SD_Lock(
    SDI_HANDLE SdiHandle,
    char *username)
```

SD_Lock performs the operations of the **AceSetUserName** and **AceLock** functions. **SD_Lock** should only be called after successfully calling **SD_Init**.

For information about using this function to help prevent unauthorized authentications, see “New Two-Step Authentication Support” on page 7.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
username	A pointer to the NULL-terminated username string. The username must contain fewer than 32 characters.

Return Values

Note: This function returns authentication result codes as return values.

ACM_OK	The username lock request has been sent to the RSA ACE/Server. Use the SD_Check function to process the PASSCODE.
ACE_UNDEFINED_USERNAME	The username argument is an invalid length or is NULL.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACM_ACCESS_DENIED	Communication with the RSA ACE/Server has failed. The failure might be the result of a communication timeout or because the node secret between the Agent and the Server is mismatched. Work with the RSA ACE/Server administrator to troubleshoot the problem.

SD_Next

Description

```
int SD_Next(  
    SDI_HANDLE SdiHandle,  
    char *nextcode)
```

SD_Next performs the Next Code operation, which takes a second successive tokencode from a user and checks its validity. **SD_Next** should be called only in response to an **ACM_NEXT_CODE_REQUIRED** return from **SD_Check**. The integrating application must perform all I/O because **SD_Next** does not display the Next Code prompt.

This function is a synchronous wrapper for **AceSetNextPasscode**, and **AceSendNextPasscode**.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
nextcode	A pointer to the NULL terminated PASSCODE string.

Return Values

Note: This function returns authentication result codes as return values.

ACM_OK	The user successfully authenticated. You can use the AceGetShell function to access the shell field.
ACE_UNDEFINED_NEXT_PASSCODE	The nextcode argument is an invalid length or is NULL.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACM_ACCESS_DENIED	The user failed authentication.

SD_Pin

Description

```
int SD_Pin(
    SDI_HANDLE SdiHandle,
    char *pin)
```

SD_Pin performs the New PIN operation in which a new PIN is transmitted to the RSA ACE/Server for storage in a token record. **SD_Pin** should be called only in response to an **ACM_NEW_PIN_REQUIRED** return from **SD_Check**. The integrating application must perform all I/O, because **SD_Pin** does not display the New PIN prompts and messages.

Note: Do not consider users to be authenticated as soon as they have completed the New PIN operation. You must require that they authenticate using the new PIN.

Input Arguments

SdiHandle	The value of a handle originally assigned by a call to SD_Init .
pin	A pointer to the NULL terminated PIN string. The PIN must contain 4 to 8 characters. If you pass a NULL pointer or an empty string, it will result in an aborted new PIN process, which is equivalent to the AceCancelPin function.

Return Values

Note: This function returns authentication result codes as return values.

ACE_INVALID_ARG	The pin argument is an invalid length.
ACE_ERR_INVALID_HANDLE	The handle value is invalid.
ACM_NEW_PIN_ACCEPTED	The RSA ACE/Server has accepted the new PIN. The user should now be required to authenticate with the new PIN.
ACM_NEW_PIN_REJECTED	The RSA ACE/Server rejected the new PIN. The PIN might not have matched the parameters set in the return from AceGetPinParams .

Appendix A: Return Values and Result Codes

This appendix identifies the RSA ACE/Agent Authentication API return values and result codes that your code must process to produce correct results.

Return Values

The following table presents the complete list of **return values**.

Return Value	Meaning
ACE_CFGFILE_NOT_FOUND	The sdconf.rec file could not be opened.
ACE_CFGFILE_READ_FAIL	The sdconf.rec file was short or unreadable.
ACE_ERR_INVALID_HANDLE	The handle passed was invalid.
ACE_ERR_NOT_INITIALIZED	A call was made to an API function before AceInitialize was called.
ACE_EVENT_CREATE_FAIL	A notification event could not be created. This return value is unusual. A systemic failure might cause this result.
ACE_INIT_NO_RESOURCE	A low memory condition caused AceInit to fail.
ACE_INIT_SOCKET_FAIL	Could not create a socket in AceInit .
ACE_INVALID_ARG	An invalid argument was supplied to an API function.
ACE_NOT_ENOUGH_STORAGE	The supplied buffer has insufficient storage for the data.
ACE_PROCESSING	This value is returned only by the asynchronous calls. The call was successful in initiating the requested asynchronous operation.
ACE_PTHREADATTR_CREATE_FAIL	The UNIX pthread library failed to create a pthread_attr structure. This return value is unusual. A systemic failure might cause this result.
ACE_PTHREADATTR_FAIL	The UNIX pthread library failed to set a thread attribute. This return value is unusual. A systemic failure might cause this result.
ACE_PTHREADCONDVAR_CREATE_FAIL	The UNIX pthread library failed to create a conditional variable. A low memory condition can cause this error.

Return Value	Meaning
ACE_PTHREAD_CREATE_FAIL	The UNIX pthread library failed to create the thread that processes the asynchronous requests. This return value is unusual. A systemic failure might cause this result.
ACE_PTHREADMUTEX_CREATE_FAIL	The UNIX pthread library failed to create a mutex . A low memory condition can cause this error.
ACE_SOCKET_LIB_NOT_FOUND	The Windows socket library winsock.dll could not be initialized or is the wrong version.
ACE_SUCCESS	The call was successful.
ACE_THREAD_CREATE_FAIL	The internal threads that control the asynchronous operations could not be created. This return value is unusual. A systemic failure can cause this result.
ACE_TOO_MANY_CALLERS	An attempt was made to use a handle while a call was already in progress. This can happen if a multi-threaded application uses the same handle from multiple threads at the same time.
ACE_UNDEFINED_CLIENTADDR	The client IP address has not been set by AceSetUserClientAddress .
ACE_UNDEFINED_NEXT_PASSCODE	The next tokencode has not been set by AceSetNextPasscode .
ACE_UNDEFINED_PASSCODE	The PASSCODE has not been set by AceSetPasscode .
ACE_UNDEFINED_PIN	The PIN has not been set by AceSetPin .
ACE_UNDEFINED_USERNAME	The user name has not been set by AceSetUsername .

Result Codes

This table presents the complete list of **result codes**.

Result Code	Meaning
ACM_ACCESS_DENIED	The RSA ACE/Server denied the authentication. That is, the user's attempt to authenticate failed.
ACM_ACK_NAMELOCK	Not returned—for internal use only.
ACE_CHECK_PIN_REQ_NOT_KNOWN	The RSA ACE/Server PIN parameter user_selectable was not in the allowed range of values.
ACM_DOWNGRADE	Not returned—for internal use only.

Result Code	Meaning
ACM_INVALID_SERVER	Used only for RSA ACE/Server v1.x. The Server node secret does not match the node secret of the RSA ACE/Agent.
ACM_LOG_ACK	Not returned—for internal use only.
ACM_NEW_PIN_ACCEPTED	The RSA ACE/Server accepted the new PIN.
ACM_NEW_PIN_REJECTED	The RSA ACE/Server rejected the new PIN.
ACM_NEW_PIN_REQUIRED	The authentication was successful, but the user must select a new PIN.
ACM_NEXT_CODE_BAD	Not returned—for internal use only.
ACM_NEXT_CODE_REQUIRED	The authentication was successful, but the user must enter the next tokencode.
ACM_NO_SERVER	The RSA ACE/Server is not responding.
ACM_OK	The meaning of this result code depends on the function that caused it to be set. Refer to the following table for details.
ACM_OK_2	Not returned—for internal use only.
ACM_SHELL_BAD	Not returned—for internal use only.
ACM_SHELL_OK	Not returned—for internal use only.
ACM_SUSPECT_ACK	Not returned—for internal use only.
ACM_TIME_OK	Not returned—for internal use only.

ACM_OK

This table explains the meaning of the result code **ACM_OK**.

Function Call that Caused ACM_OK Result Code to Be Set	Meaning of ACM_OK
AceInit SD_Init	The call was successful and required resources were allocated.
AceCheck AceClientCheck AceSendNextPasscode SD_Check SD_ClientCheck SD_Next	The authentication was successful.
AceClose SD_Close	The call was successful and resources were deallocated.
AceStartAuth AceContinueAuth AceCloseAuth	The call was successful, but no information is yet available on the authentication status.
AceLock SD_Lock	The call was successful.

Appendix B: Using Synchronous and Asynchronous API Functions

Although the internal API functions share much code, there are internal state resources that are managed by both synchronous and asynchronous calls. Mixing synchronous API function calls with asynchronous API function calls will produce unexpected results.

Important: If you are using the RSA ACE/Agent Authentication API v 4.x **sd_**xxx legacy functions, you will only need to relink your code. RSA Security *strongly* recommends, however, that you replace the **sd_**xxx functions with the new **SD_**xxx functions to take advantage of two-step authentication and multithreaded application support. Applications for the RSA Security Secured Partners certification program will not be accepted if the corresponding code fails to support two-step authentication.

The following table states how the API function calls can be used with one another.

Function Call Group	Other Function Calls That Can Be Used With the Group
The handles of the synchronous calls AceCloseAuth , AceContinueAuth , and AceStartAuth	<p>Can be used with the synchronous calls AceGetAuthenticationStatus, AceGetShell, AceGetTime, and AceGetUserData.</p> <p>Can be used with the synchronous calls AceSetTimeout and AceSetUserData.</p> <p>(The sample code file named \ACEAgentSDK\samples\sync demonstrates the AceCloseAuth, AceContinueAuth, AceGetAuthenticationStatus, and AceStartAuth calls.)</p>
The synchronous calls AceGetAlphanumeric , AceGetAuthenticationStatus , AceGetMaxPinLen , AceGetMinPinLen , AceGetPinParams , AceGetShell , AceGetSystemPin , AceGetTime , AceGetUserData , and AceGetUserSelectable	<p>Can be used with the synchronous calls SD_Check, SD_ClientCheck, SD_Close, SD_Init, SD_Lock, SD_Next, and SD_Pin.</p> <p>Can be used with the asynchronous calls AceCancelPin, AceCheck, AceClientCheck, AceClose, AceInit, AceLock, AceSendNextPasscode, and AceSendPin.</p>

Function Call Group	Other Function Calls That Can Be Used With the Group
The synchronous calls AceSetNextPasscode , AceSetPasscode , AceSetPin , AceSetTimeout , AceSetUserData , and AceSetUsername	<p>Can be used with the asynchronous calls AceCancelPin, AceCheck, AceClientCheck, AceClose, AceInit, AceLock, AceSendNextPasscode, and AceSendPin.</p> <p>Note: The AceSetTimeout and AceSetUserData functions can <i>also</i> be used with AceCloseAuth, AceContinueAuth, and AceStartAuth, as well as with the SD_xxx functions.</p> <p>(For the synchronous calls AceSetNextPasscode, AceSetPasscode, AceSetPin, and AceSetUsername, look at the sample code file named <code>\ACEAgentSDK\samples\async</code>.)</p>
The handles of the synchronous calls SD_Check , SD_ClientCheck , SD_Close , SD_Init , SD_Lock , SD_Next , and SD_Pin	<p>Can be used with the synchronous calls AceGetAlphanumeric, AceGetAuthenticationStatus, AceGetMaxPinLen, AceGetMinPinLen, AceGetPinParams, AceGetShell, AceGetSystemPin, AceGetTime, AceGetUserData, and AceGetUserSelectable.</p> <p>Can be used with the synchronous calls AceSetTimeout and AceSetUserData.</p> <p>(The sample code file named <code>\ACEAgentSDK\samples\sync2</code> demonstrates the SD_Check, SD_Close, SD_Init, SD_Lock, SD_Next, and SD_Pin calls.)</p>
The asynchronous calls AceCancelPin , AceCheck , AceClientCheck , AceClose , AceInit , AceLock , AceSendNextPasscode , and AceSendPin	<p>Can be used with the synchronous calls AceGetAlphanumeric, AceGetAuthenticationStatus, AceGetMaxPinLen, AceGetMinPinLen, AceGetPinParams, AceGetShell, AceGetSystemPin, AceGetTime, AceGetUserData, and AceGetUserSelectable.</p> <p>Can be used with the synchronous calls AceSetNextPasscode, AceSetPasscode, AceSetPin, and AceSetUsername.</p> <p>(The sample code file named <code>\ACEAgentSDK\samples\async</code> demonstrates the AceCancelPin, AceCheck, AceClientCheck, AceClose, AceInit, AceLock, AceSendNextPasscode, and AceSendPin calls.)</p>
<p>Note: AceInitialize is called once before calling any other function. AceCleanup and AceShutdown will work with any other function, but they operate only when needed to cancel authentications, close sockets, and terminate processing. Your code can call AceCleanup as many times as you like. Only make a call to AceShutdown when you want to terminate all processing.</p>	

Appendix C: Modifying the Message Catalog

The API functions **AceStartAuth** and **AceContinueAuth** return to the calling application a set of prompts that must be displayed to the user. The strings for these prompts are stored in a message catalog that you can customize. For example, the messages can be modified if it is necessary to display these strings in languages other than English. The same catalog contains all of the messages that the API writes to the UNIX syslog and Windows NT Event Log.

To customize the strings on UNIX, you must have a passing familiarity with the UNIX **gencat** facility. To modify the strings on Windows NT, you must have and be able to use the Microsoft Message Compiler (part of Microsoft Developer Studio 6.0). Modification of the message catalog cannot be delegated to an RSA ACE/Server or RSA ACE/Agent administrator who does not have development capabilities.

Customizing Message Strings in UNIX

On each of the UNIX platforms supported, the file **ACEAgentSDK/src/sdmsg.msg** is used as the source to create a file named **sdmsg.cat**, which in turn is used by the API library to generate the messages that are logged in the syslog facility and returned as prompts through **AceStartAuth** and **AceContinueAuth**. The messages used by these API entry points are indexed starting at number 1100. You can edit the **sdmsg.msg** file to supply messages in a different language or with alternate wording. You can modify the text of existing messages, but you *cannot* add additional messages to this file.

After you have modified the **sdmsg.msg** file, you can “compile” the file into the **sdmsg.cat** file by using the following UNIX command:

```
gencat sdmsg.cat sdmsg.msg
```

The **sdmsg.cat** file must be stored in the same directory as the **sdconf.rec** configuration file.

Customizing Message Strings in Windows NT

To accomplish the same result on Windows NT, the file **ACEAgentSDK/src/sdmsg.mc** is provided. The content of **sdmsg.mc** is similar to the content of the **sdmsg.msg** file for UNIX platforms. The messages for the **AceStartAuth** and **AceContinueAuth** API entry points are listed starting at index number 1100. The **sdmsg.mc** file must be compiled with the Microsoft Message Compiler utility MC to produce a binary file named **MSG00001.BIN**. To change the messages for Windows NT, you must modify the **SDMSG.DLL** file installed in the **%SystemRoot%\System32** directory.

To customize the message strings:

1. Compile **sdmsg.mc** with Microsoft Message Compiler using this command line:

```
mc sdmsg.mc
```

This compilation produces the binary file **MSG00001.BIN**.

2. Create a copy of the **SDMSG.DLL** file, and open it with Microsoft Developer Studio 6.0. Use the **Open as Resources** option to allow editing of the resources contained in the file.
3. Open the **MSG00001.BIN** file. Select the entire contents of the file shown as binary data, and copy it to the Clipboard using the **Copy** command on the Edit menu.
4. In the **SDMSG.DLL** file, locate the resource of type “11” with ID 1. Open the resource with ID 1 and select the entire contents.
5. Paste the new contents from the Clipboard into **SDMSG.DLL** by using the **Paste** command on the Edit menu.
6. Modify the version resource description information of the **SDMSG.DLL** file to indicate that it is a customized file.
7. Save the **SDMSG.DLL** file, and copy it to the **%SystemRoot%\System32** directory.

Appendix D: Legacy Functions

RSA ACE/Agent Authentication API version 5.0 will allow you to reuse your existing Agent code by relinking with the current API libraries. The following legacy functions are still supported in version 5.0:

- **creadcfg**
- **sd_check**
- **sd_close**
- **sd_init**
- **sd_next**
- **sd_pin**

In future versions of the API, **creadcfg** will no longer be supported. It has been replaced in version 5.0 by **AceInitialize**. Refer to the **AceInitialize** description in this guide for more information.

In future versions of the API, **sd_init**, **sd_check**, **sd_pin**, **sd_next**, and **sd_close** will no longer be supported. These functions have been superseded by the new thread-safe functions **SD_Init**, **SD_Check**, **SD_Pin**, **SD_Next**, and **SD_Close**. The legacy **sd_xxx** functions had direct access to data structures. The new **SD_xxx** functions use the **AceSetxxx** and **AceGetxxx** functions to operate on data structures.

Important: If you are using the RSA ACE/Agent Authentication API v 4.x **sd_xxx** legacy functions, you will only need to relink your code to use the new API libraries. RSA Security *strongly* recommends, however, that you replace the **sd_xxx** functions with the new **SD_xxx** functions to take advantage of two-step authentication and multithreaded application support. Applications for the RSA Security Secured Partners certification program will not be accepted if the corresponding code fails to support two-step authentication.

Index

A

- AceCancelPin, 24
- AceCheck, 26
- AceCleanup, 10, 28
- AceClientCheck, 29
- AceClose, 31
- AceCloseAuth, 8, 33
- AceContinueAuth, 8, 10, 34
- AceGetAlphanumeric, 8, 36
- AceGetAuthenticationStatus, 8, 9, 37
- AceGetMaxPinLen, 8, 38
- AceGetMinPinLen, 8, 39
- AceGetPinParams, 10, 40
- AceGetShell, 8, 41
- AceGetSystemPin, 8, 42
- AceGetTime, 8, 43
- AceGetUserData, 8, 44
- AceGetUserSelectable, 8, 45
- AceInit, 46
- AceInitialize, 9, 10, 48, 81
- AceLock, 7, 10, 49
- AceSendNextPasscode, 51
- AceSendPin, 53
- AceSetNextPasscode, 55
- AceSetPasscode, 56
- AceSetPin, 57
- AceSetTimeout, 10, 58
- AceSetUserClientAddress, 59
- AceSetUserData, 10, 60
- AceSetUsername, 61
- AceShutdown, 10, 11, 62
- AceStartAuth, 8, 10, 63
- acexport.h, 20
- ACM_OK table, 75
- Asynchronous
 - callbacks, 9
 - list of functions, 20
 - operations, 20
 - sample code, 6
 - usage recommendations, 77
- Authentication, support for two-step version, 7

C

- Callbacks
 - asynchronous, 9
 - cleanup, 10
- Categories of API functions, 18
- Compatibility with previous versions, 7
- creadcfg, 48, 81
- Customer service information, 6

D

- Data encapsulation, 8
- Developer profile, 5
- Dynamic UNIX libraries, 21

E

- Encapsulating data, 8
- Expiration facility, 10

F

- Failover functionality, 8
- Function categories, 18

I

- Interoperating with previous versions, 7

L

- Legacy functions, 81
- libaceclnt.a, 10
- Load balancing, 8
- Log messages, 10, 79

M

- Message catalog, 10, 79
- Message prompts, 10
 - modifying, 79

N

- New functions, 10
- New PIN Mode, handling, 17
- Next Tokencode Mode, handling, 17
- Normal authentication, handling, 17

P

- Platform dependencies, 20
- Printing PDF files, 6
- Prompts, 10
 - modifying, 79

R

- Result codes table, 74
- Return values table, 73
- RSA ACE/Agent description, 14
- RSA ACE/Server versions, supported, 7

S

- Sample code, 6
- SD_Check, 11, 65
- SD_ClientCheck, 11, 66
- SD_Close, 11, 67
- SD_Init, 11, 68
- SD_Lock, 7, 11, 69
- SD_Next, 11, 70
- SD_Pin, 11, 71
- sdconf.rec, 8
- SDI_HANDLE, design constraint on use of, 9
- sdclient.a, name change of, 10
- sdopts.rec, 8
- Service and support information, 6
- Static UNIX libraries, 21
- Supported RSA ACE/Server versions, 7
- Synchronous
 - function sets, 9
 - sample code, 6
 - usage recommendations, 77

T

- Taxonomy of API functions, 18
- Technical support, 6
- Thread safety, 8

- Two-factor authentication, 13
- Two-step authentication support, 7

U

UNIX

- AceInitialize, 9
- compiler option, 20
- dynamic and static libraries, 21
- library name change, 10
- sample code, 6

- Updated functions, 10
- Using synchronous and asynchronous API functions, 77

W

Windows NT

- AceInitialize, 9, 48
- compiler option, 20
- sample code, 6