

UniVerse

Security Features

Version 10.2
September, 2006

IBM Corporation
555 Bailey Avenue
San Jose, CA 95141

Licensed Materials – Property of IBM

© Copyright International Business Machines Corporation 2006. All rights reserved.

AIX, DB2, DB2 Universal Database, Distributed Relational Database Architecture, NUMA-Q, OS/2, OS/390, and OS/400, IBM Informix®, C-ISAM®, Foundation.2000™, IBM Informix® 4GL, IBM Informix® DataBlade® module, Client SDK™, Cloudscape™, Cloudsync™, IBM Informix® Connect, IBM Informix® Driver for JDBC, Dynamic Connect™, IBM Informix® Dynamic Scalable Architecture™ (DSA), IBM Informix® Dynamic Server™, IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager), IBM Informix® Extended Parallel Server™, i.Financial Services™, J/Foundation™, MaxConnect™, Object Translator™, Red Brick® Decision Server™, IBM Informix® SE, IBM Informix® SQL, InformiXML™, RedBack®, SystemBuilder™, U2™, UniData®, UniVerse®, wIntegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

This product includes cryptographic software written by Eric Young (eay@cryptosoft.com).

This product includes software written by Tim Hudson (tjh@cryptosoft.com).

Documentation Writer: Claire Gustafson, Shelley Thompson

US GOVERNMENT USERS RESTRICTED RIGHTS

Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of Contents

Preface

Organization of This Manual	ii
Documentation Conventions.	iii
UniVerse Documentation.	v
Related Documentation	viii
API Documentation	ix

Chapter 1

Configuring SSL Through UniAdmin

Configuring SSL Through UniAdmin.	1-2
Accessing UniVerse SSL Configuration Dialog Box	1-3
Creating a Certificate Request	1-4
Creating a Certificate	1-11
Creating a Security Context	1-19
Configuring SSL for UniObjects for Java or Telnet	1-35

Chapter 2

Using SSL with the CallHTTP and Socket Interfaces

Overview of SSL Technology	2-3
Setup and Configuration for SSL	2-4
SSL Security Programmatic Interfaces for UniData and UniVerse	2-5
Creating A Security Context.	2-6
Saving a Security Context	2-8
Loading a Security Context	2-10
Showing a Security Context	2-12
Adding a Certificate	2-13
Adding an Authentication Rule	2-16
Setting a Cipher Suite	2-18
Getting A Cipher Suite	2-26
Setting a Private Key	2-28
Setting Client Authentication Mode	2-31
Setting the Authentication Depth	2-32
Generating a Key Pair.	2-34

Creating a Certificate Request	2-37
Creating a Certificate	2-41
Setting a Random Seed	2-43
Analyzing a Certificate	2-45
Encoding and Cryptographic Functions	2-46
Encoding Data.	2-47
Encrypting Data	2-49
Generating a Message Digest	2-55
Generating a Digital Signature.	2-57
Additional Reading	2-60

Chapter 3 Using SSL With UniObjects for Java

Overview of SSL Technology	3-3
Software Requirements	3-4
Setting up Java Secure Socket Extension (JSSE)	3-5
Configuring UOJ to use IBM JSSE	3-6
Configuring the Database Server for SSL	3-7
Creating a Secure Connection	3-9
Direct Connection	3-10
Establishing the Connection	3-12
Proxy Tunneling	3-13
Externally Secure	3-15
Managing Keys and Certificates for a UOJ Client and a Proxy Server	3-20
Importing CA Certificates Into UOJ Client Trustfile	3-20
Generating client certificates.	3-21
Managing Keyfile and Trustfile for the Proxy Server.	3-22

Chapter 4 Automatic Data Encryption

Encrypted File Types	4-4
Encryption With UniVerse Replication	4-4
Key Store	4-5
How Encryption Works	4-6
Defining a Master Key	4-8
Changing a Master Key After Data is Encrypted	4-8
UniVerse Encryption Algorithms	4-9
Encryption Commands	4-10
CREATE.ENCRYPTION.KEY	4-10
DELETE.ENCRYPTION.KEY	4-10
LIST.ENCRYPTION.KEY	4-11
GRANT.ENCRYPTION.KEY	4-11
REVOKE.ENCRYPTION.KEY	4-12

ENCRYPT.FILE	4-13
DECRYPT.FILE	4-17
LIST.ENCRYPTION.FILE.	4-22
ACTIVATE.ENCRYPTION.KEY	4-22
DEACTIVATE.ENCRYPTION.KEY	4-23
DISABLE.DECRYPTION	4-23
ENABLE.ENCRYPTION	4-24
UniVerse BASIC Encryption Commands	4-25
ACTIVATEKEY	4-25
DEACTIVATEKEY	4-25
DISABLEDEC	4-26
ENABLEDEC	4-27
STATUS Function Changes	4-27
The encman Utility.	4-29
Viewing Audit Trail Information	4-29
Generating a Key Store	4-30
Deleting the Key Store	4-31

Chapter 5 **Using UniAdmin for Data Encryption**

Using UniAdmin for Encryption	5-2
Adding an Encryption Key.	5-3
Deleting an Encryption Key	5-3
Viewing Encryption Key Details	5-4
Granting Privileges	5-5
Revoking Privileges	5-6
Encrypting a File	5-7
Decrypting a File	5-10
Listing Encryption Information	5-13
Viewing Audit Information	5-15

Preface

This manual is for application developers and system administrators who want to learn how to use implement SSL security.

Organization of This Manual

This manual contains the following:

Chapter 1, “[Configuring SSL Through UniAdmin](#),” describes how to configure SSL using UniAdmin.

Chapter 2, “[Using SSL with the CallHTTP and Socket Interfaces](#),” describes how to set up and configure SSL for use with the CallHTTP and Socket interfaces.

Chapter 3, “[Using SSL With UniObjects for Java](#),” explains how to use SSL (Secure Socket Layer) with UniObjects for Java (UOJ).

Chapter 4, “[Automatic Data Encryption](#),” describes how you can encrypt specified fields or entire records, and automatically decrypt data when accessed by UniVerse or UniVerse BASIC commands.

Chapter 5, “[Using UniAdmin for Data Encryption](#),” describes how use UniAdmin to manage data encryption on your system.

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates UniVerse identifiers such as file names, account names, schema names, and Windows NT file names and paths.
<i>Italic</i>	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, file names, and paths.
Courier	Courier indicates examples of source code and system output.
Courier Bold	In examples, courier bold indicates characters that the user types or keys the user presses (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
ä	A right arrow between menu options indicates you should choose each option in sequence. For example, “Choose File ä Exit ” means you should choose File from the menu bar, then choose Exit from the File pull-down menu.
⌘	Item mark. For example, the item mark (⌘) in the following string delimits elements 1 and 2, and elements 3 and 4: 1⌘2F3⌘4V5

Documentation Conventions

Convention	Usage
F	Field mark. For example, the field mark (F) in the following string delimits elements FLD1 and VAL1: FLD1 F VAL1 V SUBV1 S SUBV2
V	Value mark. For example, the value mark (V) in the following string delimits elements VAL1 and SUBV1: FLD1 F VAL1 V SUBV1 S SUBV2
S	Subvalue mark. For example, the subvalue mark (S) in the following string delimits elements SUBV1 and SUBV2: FLD1 F VAL1 V SUBV1 S SUBV2
T	Text mark. For example, the text mark (T) in the following string delimits elements 4 and 5: 1 F 2 S 3 V 4 T 5

Documentation Conventions (Continued)

The following are also used:

- ⁿ Syntax definitions and examples are indented for ease in reading.
- ⁿ All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- ⁿ Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

UniVerse Documentation

UniVerse documentation includes the following:

UniVerse Installation Guide: Contains instructions for installing UniVerse 10.2.

UniVerse New Features Version 10.2: Describes enhancements and changes made in the UniVerse 10.2 release for all UniVerse products.

UniVerse BASIC: Contains comprehensive information about the UniVerse BASIC language. It includes reference pages for all UniVerse BASIC statements and functions. It is for experienced programmers.

UniVerse BASIC Commands Reference: Provides syntax, descriptions, and examples of all UniVerse BASIC commands and functions.

UniVerse BASIC Extensions: Describes the following extensions to UniVerse BASIC: UniVerse BASIC Socket API, Using CallHTTP, and Using WebSphere MQ with UniVerse.

UniVerse BASIC SQL Client Interface Guide: Describes how to use the BASIC SQL Client Interface (BCI), an interface to UniVerse and non-UniVerse databases from UniVerse BASIC. The BASIC SQL Client Interface uses ODBC-like function calls to execute SQL statements on local or remote database servers such as UniVerse, DB2, SYBASE, or INFORMIX. This book is for experienced SQL programmers.

Administering UniVerse: Describes tasks performed by UniVerse administrators, such as starting up and shutting down the system, system configuration and maintenance, system security, maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, and network services. This book includes descriptions of how to use the UniVerse Admin program on a Windows client and how to use shell commands on UNIX systems to administer UniVerse.

Using UniAdmin: Describes the UniAdmin tool, which enables you to configure UniVerse, configure and manage servers and databases, and monitor UniVerse performance and locks.

UniVerse Security Features: Describes security features in UniVerse, including configuring SSL through UniAdmin, using SSL with the CallHttp and Socket interfaces, using SSL with UniObjects for Java, and automatic data encryption.

UniVerse Transaction Logging and Recovery: Describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

UniVerse System Description: Provides detailed and advanced information about UniVerse features and capabilities for experienced users. This book describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

UniVerse User Reference: Contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

Guide to Retrieve: Describes Retrieve, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

Guide to ProVerb: Describes ProVerb, a UniVerse processor used by application developers to execute prestored procedures called procs. This book describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

Guide to the UniVerse Editor: Describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. This book also includes reference pages for all UniVerse Editor commands.

UniVerse NLS Guide: Describes how to use and manage UniVerse's National Language Support (NLS). This book is for users, programmers, and administrators.

UniVerse SQL Administration for DBAs: Describes administrative tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases. This book is for database administrators (DBAs) who are familiar with UniVerse.

UniVerse SQL User Guide: Describes how to use SQL functionality in UniVerse applications. This book is for application developers who are familiar with UniVerse.

UniVerse SQL Reference: Contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

Related Documentation

The following documentation is also available:

UniVerse GCI Guide: Describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. This book is for experienced programmers who are familiar with UniVerse.

UniVerse ODBC Guide: Describes how to install and configure a UniVerse ODBC server on a UniVerse host system. It also describes how to use UniVerse ODBC Config and how to install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

UV/Net II Guide: Describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. This book is for experienced UniVerse administrators.

UniVerse Guide for Pick Users: Describes UniVerse for new UniVerse users familiar with Pick-based systems.

Moving to UniVerse from PI/open: Describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. This book includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

API Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

Administrative Supplement for APIs: Introduces IBM's seven common APIs, and provides important information that developers using any of the common APIs will need. It includes information about the UniRPC, the UCI Config Editor, the *ud_database* file, and device licensing.

UCI Developer's Guide: Describes how to use UCI (Uni Call Interface), an interface to UniVerse and UniData databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse and UniData servers. This book is for experienced SQL programmers.

IBM JDBC Driver for UniData and UniVerse: Describes UniJDBC, an interface to UniData and UniVerse databases from JDBC applications. This book is for experienced programmers and application developers who are familiar with UniData and UniVerse, Java, JDBC, and who want to write JDBC applications that access these databases.

InterCall Developer's Guide: Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

UniObjects Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

UniObjects for Java Developer's Guide: Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

UniObjects for .NET Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from .NET. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with .NET, and who want to write .NET programs that access these databases.

Using UniOLEDB: Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

Configuring SSL Through UniAdmin

Configuring SSL Through UniAdmin	1-2
Accessing UniVerse SSL Configuration Dialog Box.	1-3
Creating a Certificate Request	1-4
Creating a Certificate	1-11
Creating a Security Context	1-19
Configuring SSL for UniObjects for Java or Telnet	1-35

Configuring SSL Through UniAdmin

Secure Sockets Layer (SSL) is a transport layer protocol that provides a secure channel between two communicating programs over which you can send arbitrary application data securely. It is by far the most widely deployed security protocol used on the World Wide Web.

SSL provides server authentication, encryption, and message integrity. It can also support client authentication.

UniData currently supports CallHTTP and the Sockets API. SSL support is important for both of these protocols in order to deploy commercial applications and securely process sensitive data, such as credit card transactions.

This chapter assumes that users who want to use SSL have a basic knowledge of public key cryptography.

Accessing UniVerse SSL Configuration Dialog Box

Use the **UniVerse SSL Configuration** dialog box to administer SSL.

Select one of the following methods to access the **UniVerse SSL Configuration** dialog box:

- From the **UniAdmin** window, double-click **SSL Configure**.
- From the **UniAdmin** menu, select **Admin**, then click **SSL Configure**.

A dialog box similar to the following example appears:



Creating a Certificate Request

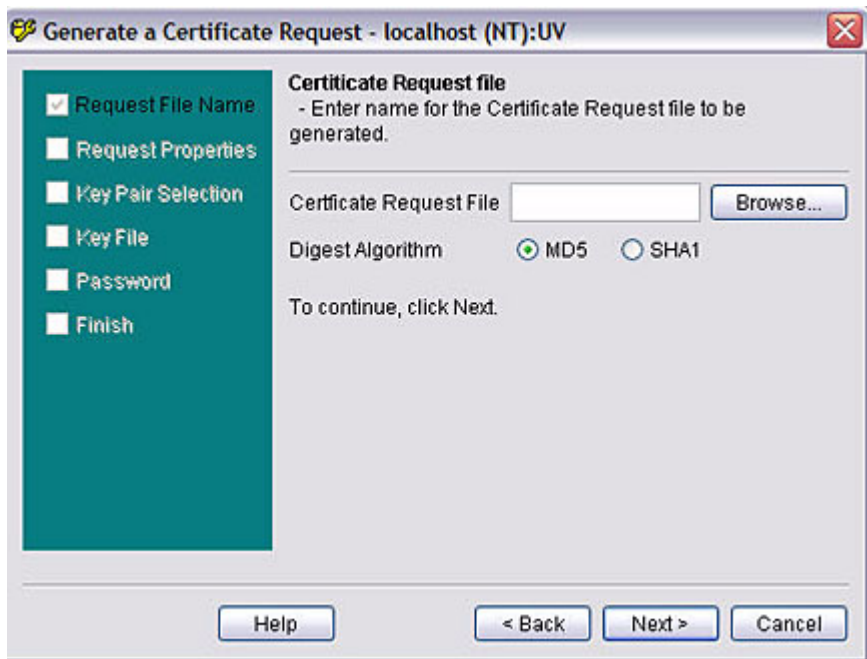
Complete the following steps to generate an X.509 certificate request, which you can send to a third-party CA to obtain a certificate, or use as input to the **Generate Certificate** wizard.

1. Click **Generate a Certificate Request**. A dialog box similar to the following example appears:



Click **Next**.

2. The **Certificate Request file** dialog box appears, as shown in the following example:



In the **Certificate Request File** box, enter the full path of the operating system-level file to hold the certificate request, or click **Browse** to search for the location.

Click the type of Digest Algorithm for the certificate request. The following types of algorithms are available:

- MD5 – MD5 hash function
- SHA1 – SHA1 hash function

Click **Next**.

3. Next, choose the properties for the certificate from the **Request Properties** dialog box, as shown in the following example:

Generate a Certificate Request - localhost (NT):UV

☒ Request File Name

☒ Request Properties

☐ Key Pair Selection

☐ Key File

☐ Password

☐ Finish

Request Properties
- Specify the request properties.

C (Country Code)

ST (Province) (optional)

L (Locality) (optional)

O (Organization)

OU (Organization Unit) (optional)

CN (Common Name)

Email (optional)

To continue, click Next.

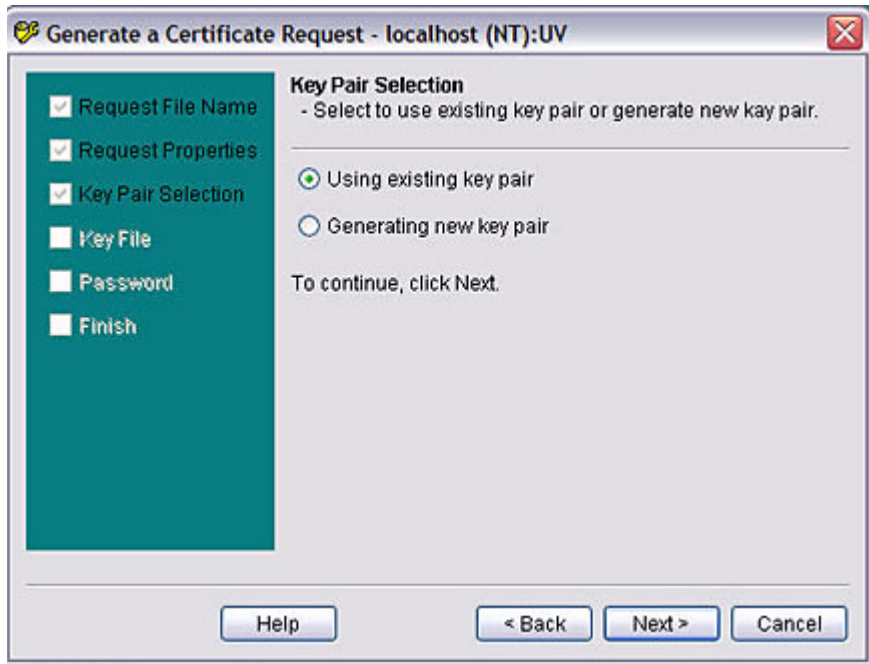
Help < Back Next > Cancel

The available properties are:

- C – Country Code
- ST – State or Province
- L – Locality (city)
- O – Organization
- OU – Organization Unit
- CN – Common Name
- Email – Email address

You must define the Country Code, Organization, and Common Name.
Click **Next**.

4. From the **Key Pair Selection** dialog box, select the type of Key Pair.



If you are using a previously generated key pair for the certificate request, select **Using existing key pair**. If you are creating a new key pair, select **Generating new key pair**.

Click **Next** to continue.

5. The **Key Pair Info** dialog box appears, as shown in the following example:

Generate a Certificate Request - localhost (NT):UV

Key Pair Info
- Select key algorithm and key file.

☒ Request File Name
☒ Request Properties
☒ Key Pair Selection
☒ Key File
☐ Password
☐ Finish

Key Algorithm: ☒ RSA ☐ DSA
Key Length: 1024
Key File Format: ☒ PEM ☐ DER
Parameter File: Browse...
Private Key File: Browse...
Public Key File: Browse...
Parameter file for DSA can be optional.
To continue, click Next.

Help < Back Next > Cancel

Specify the **Key Algorithm**. Select **RSA** if you want to use an RSA key algorithm, or **DSA** if you want to use a DSA key algorithm.

Select the key length from the **Key Length** list. Valid values range from 512 to 2048.

Select the Key File Format. Select **PEM** for a Privacy Enhanced Mail format, or **DER** for a Distinguished Encoding Rules format.

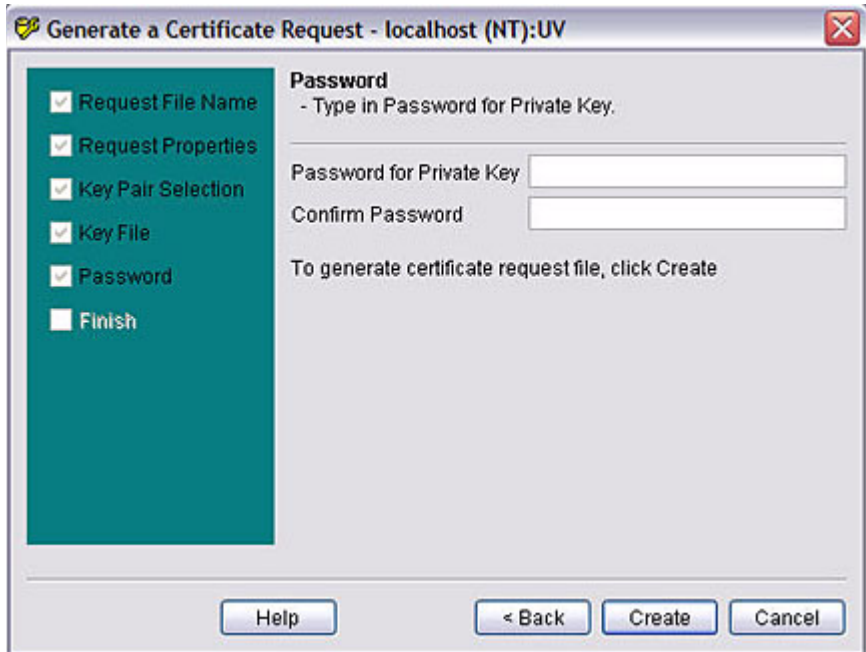
If you selected a Key File Format of DER, enter the path to the parameter file in the **Parameter File** box. UniVerse uses this file to generate a new key pair. If you leave this box empty, UniVerse uses the default.

In the **Private Key File** box, enter the name of the file in which you want to store the generated private key, or click **Browse** to search for the existing key if you selected **Use Existing Key Pair**.

In the **Public Key File** box, enter the name of the file in which you want to store the generated public key, or click **Browse** to search for the existing key if you selected **Use Existing Key Pair**.

Click **Next** to continue.

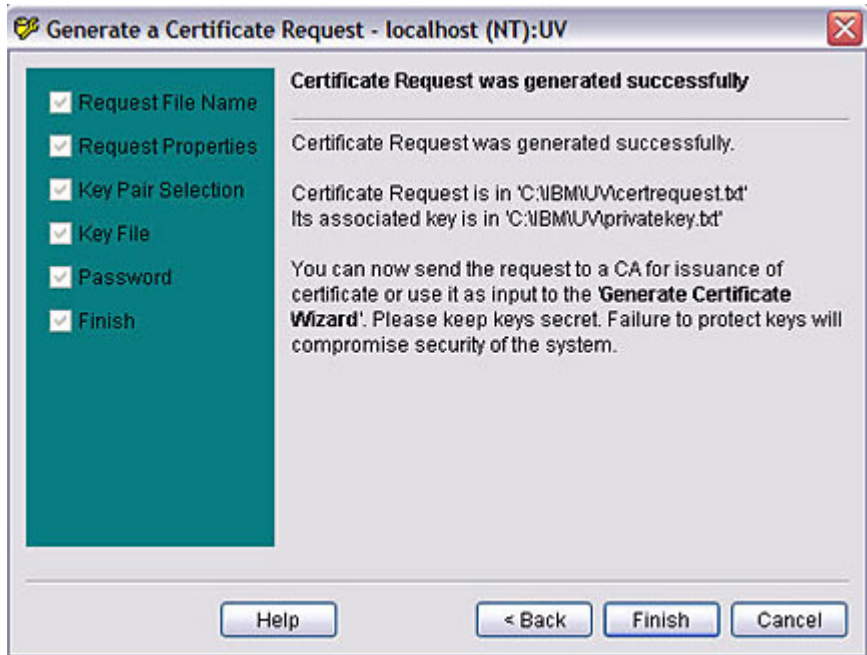
6. Next, define a password for the private key from the **Password** dialog box, as shown in the following example:



The screenshot shows a Windows-style dialog box titled "Generate a Certificate Request - localhost (NT):UV". On the left is a teal sidebar with a list of steps: "Request File Name", "Request Properties", "Key Pair Selection", "Key File", "Password", and "Finish". The "Password" step is currently selected. The main area of the dialog is titled "Password" with the instruction "- Type in Password for Private Key." Below this are two text input fields: "Password for Private Key" and "Confirm Password". A line of text below the fields says "To generate certificate request file, click Create". At the bottom of the dialog are four buttons: "Help", "< Back", "Create", and "Cancel".

In the **Password for Private Key** box, enter a password for the private key. Confirm the password by reentering it in the **Confirm Password** box.

7. Click **Create** to generate the certificate request file. The following dialog box appears after the certificate request is successfully generated:



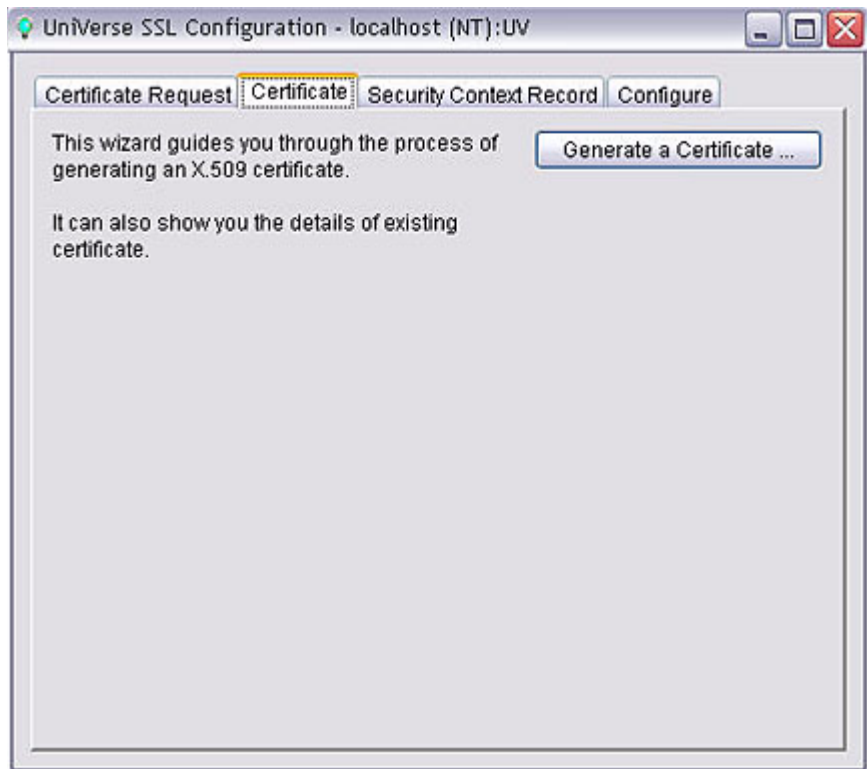
Click **Finish** to return to the **UniVerse SSL Configuration** wizard.

Creating a Certificate

You can create two types of certificates from UniAdmin:

- Self-signed certificates as a root CA that can be used later to sign other certificates.
- CA signed certificates.

To create a certificate, from the **UniVerse SSL Configuration** dialog box, click **Certificate**. A dialog box similar to the following example appears:

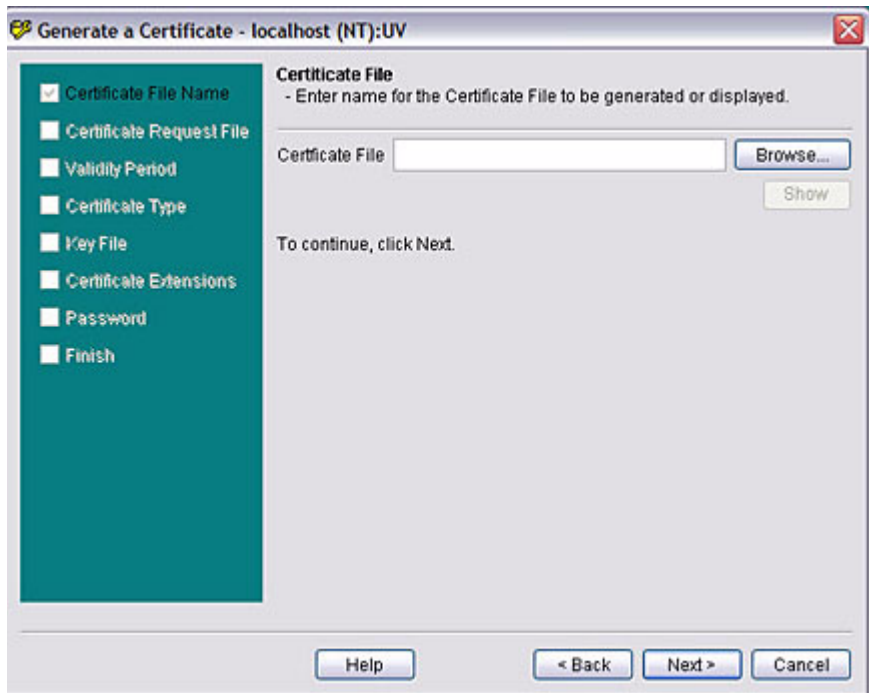


To begin generating an X.509 certificate, or to view details of an existing certificate, click **Generate a Certificate**. A dialog box similar to the following example appears:



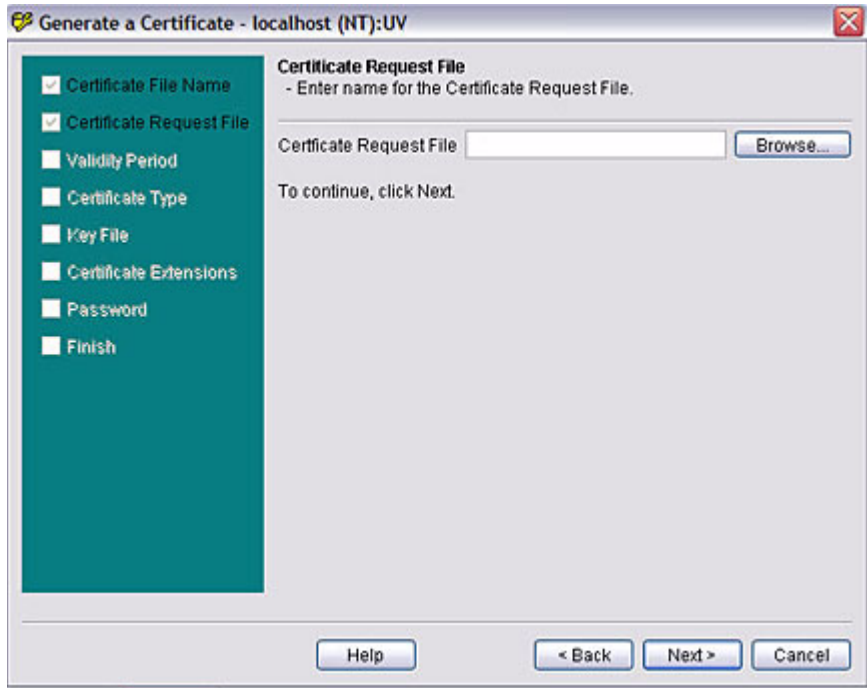
Click **Next**, then complete the following steps to create a certificate:

1. The **Certificate File** dialog box appears, as shown in the following example:



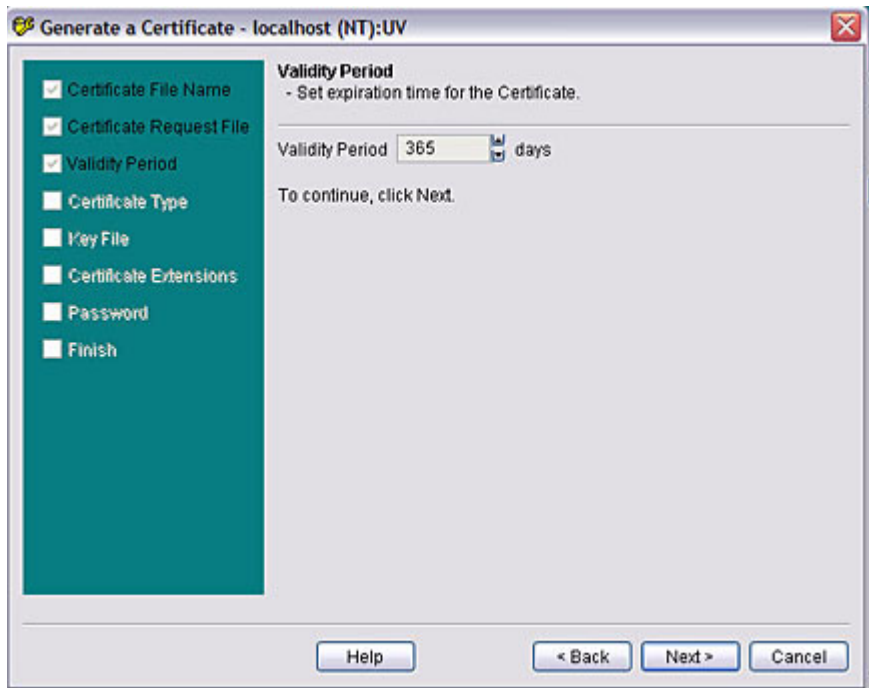
In the **Certificate File** box, enter the certificate file name, or click **Browse** search for the file. Click **Next**.

2. The **Certificate Request File** dialog box appears, as shown in the following example:



In the **Certificate Request File** box, enter the name of the file to write the generated certificate, or click **Browse** to search for the file. Click **Next**.

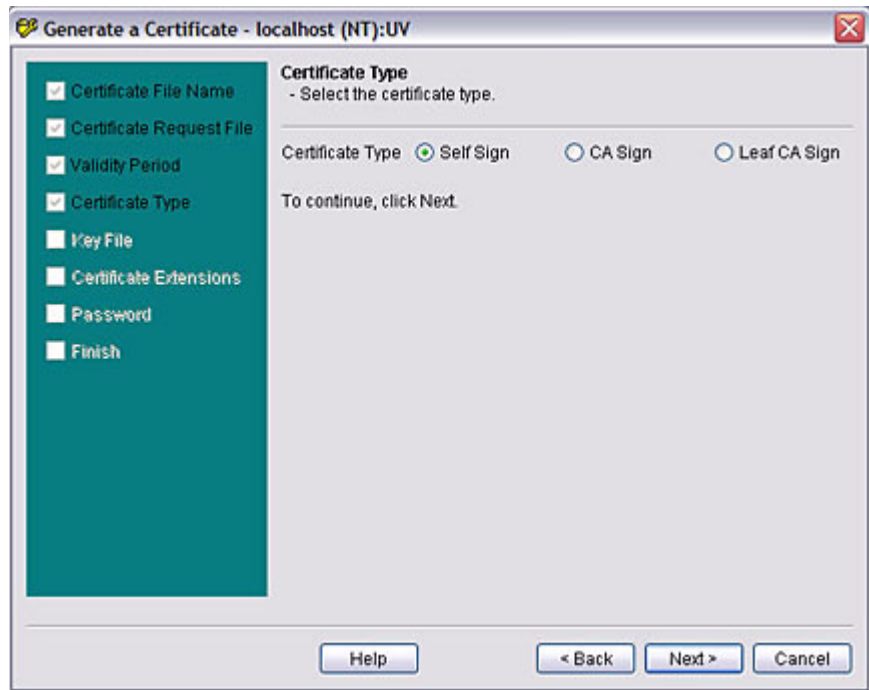
3. The **Validity Period** dialog box appears, as shown in the following example:



Select the number of days for which certificate is valid from the **Validity Period** list. The certificate is valid starting from the current date until the number of days you specify expires. The default value is 365 days.

Click **Next**.

4. The **Certificate Type** dialog box appears, as shown in the following example:

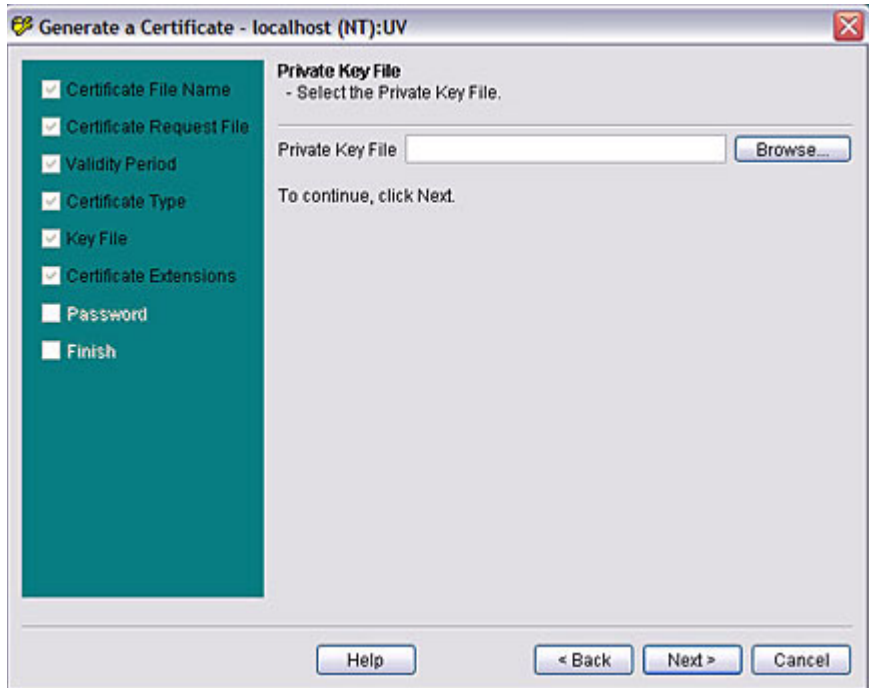


Select the type of certificate. The following types of certificates are available:

- Self Sign
- CA Sign
- Leaf CA Sign

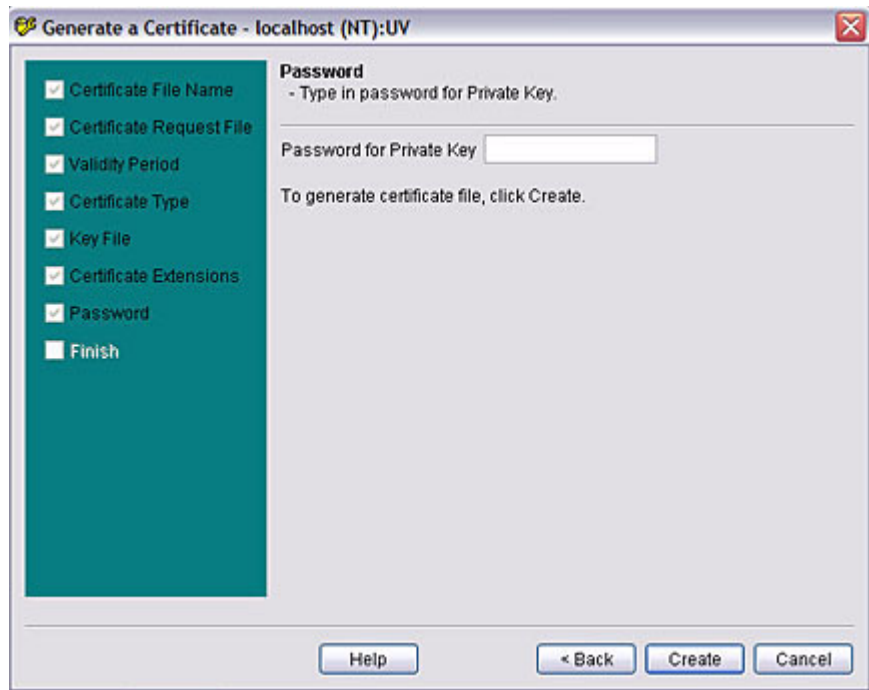
Click **Next** to continue.

5. If the type of certificate is CA or Leaf CA, you must specify a private key file associated with the signing CA certificate. Define the private key file from **Private Key File** dialog box, as shown in the following example:



Enter the name of the private key file in the **Private Key File** box, or click **Browse** to search for this file, then click **Next**.

6. Enter the password for the private key file in the **Password** dialog box, as shown in the following example:

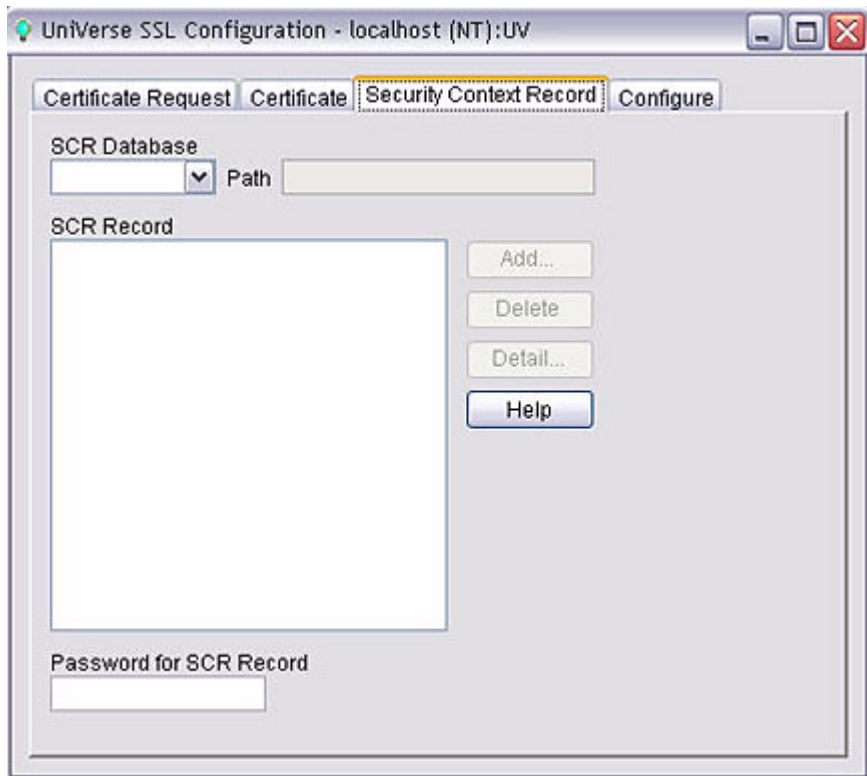


7. Click **Create** to create the certificate.

Creating a Security Context

A security context is a data structure that holds all aspects of security characteristics that the application intends to associate with a secured connection.

To create a security context, from the **UniVerse SSL Configuration** dialog box, click the **Security Context Record** tab. A dialog box similar to the following example appears:



Complete the following steps to create a Security Context Record:

1. Select the account where you want to create or view the security context record (SCR) from the **SCR Database** list. UniAdmin populates the **Path** box with the full path of the database.

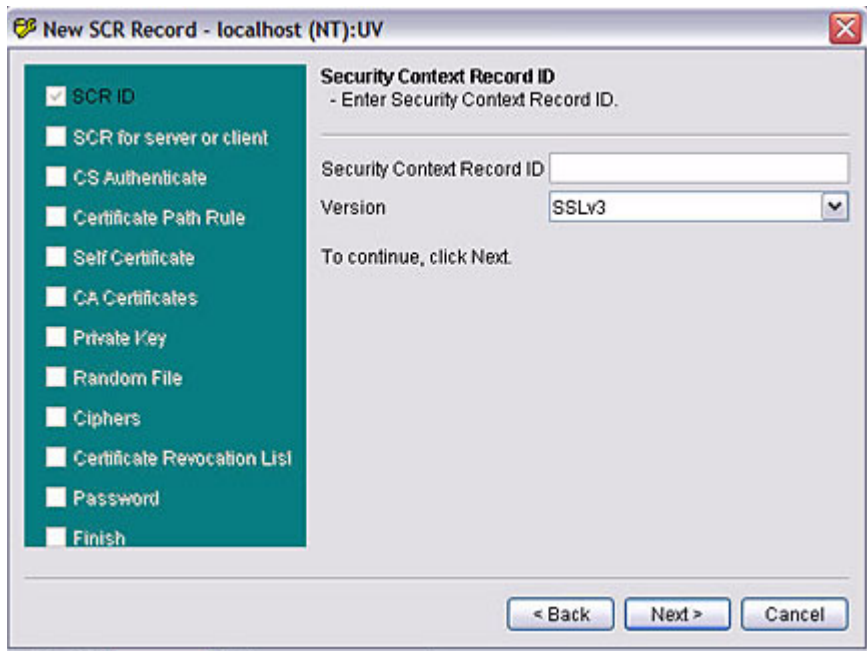
2. To add an SCR record, click **Add**. A dialog box similar to the following example appears:



Make sure you have generated the necessary keys and certificates needed before proceeding.

Click **Next** to continue.

3. The **Security Context Record ID** dialog box appears, as shown in the following example:



Enter an ID for the SCR in the **Security Context Record ID** box.

Select the appropriate version for the SCR record in the **Version** box. Valid versions are:

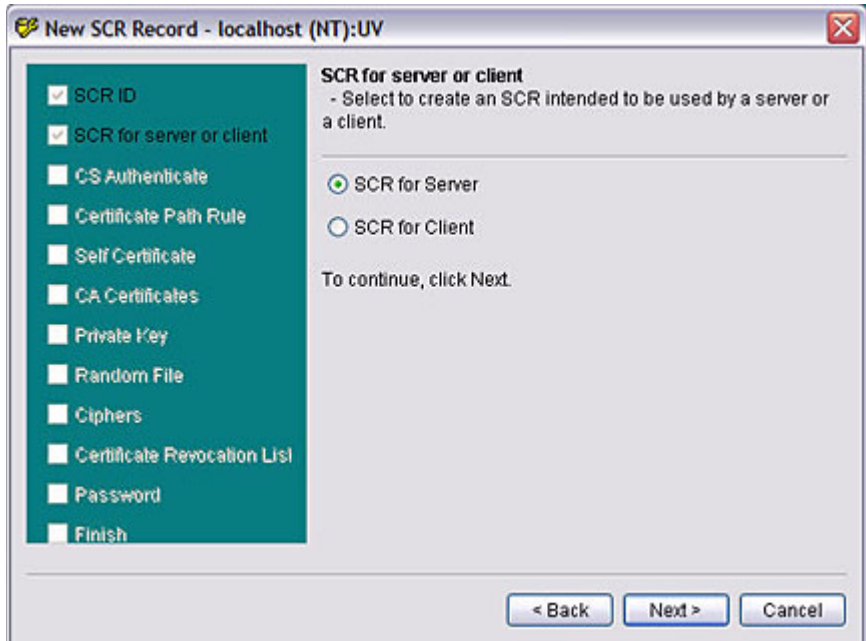
- SSLv2
- SSLv3
- TLSv1

***Note:** IBM recommends that you only use SSLv3 or TLSv1.*

Click **Next** to continue.



4. Next, define if the SCR for the server or client from the **SCR for Server or Client** dialog box, as shown in the following example:

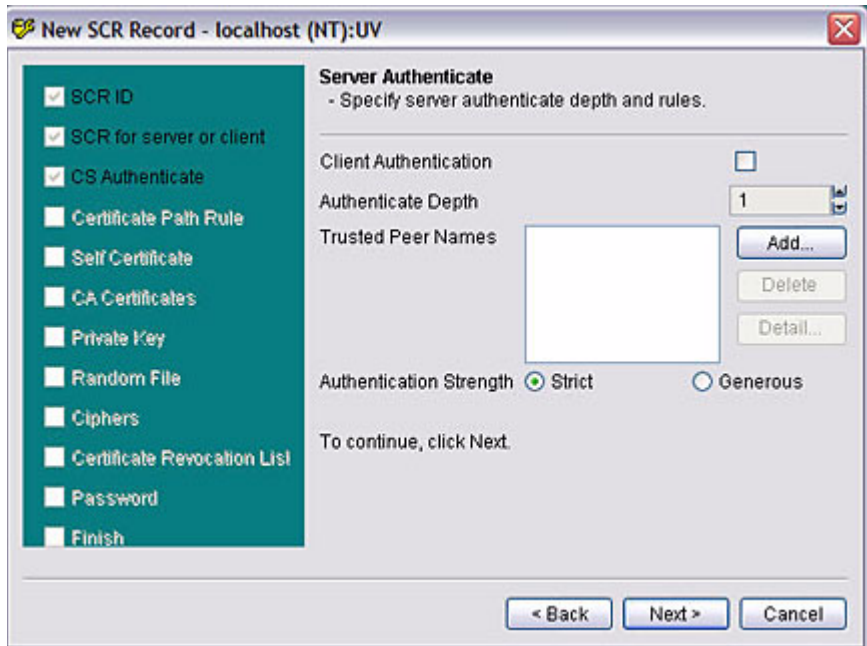


If the SCR is for use by a server, select **SCR for Server**. If the SCR is for use by a client, select **SCR for Client**.

Click **Next** to continue.

5. When you select Client Authentication, during the initial SSL handshake, the server sends the client authentication request to the client. It also receives the client certificate and performs authentication according to the issuer's certificate (or certificate chain) set in the security context.

Set authentication parameters from the **Server Authenticate** dialog box, as shown in the following example:



If you want to set authentication, select the **Client Authentication** check box.

The **Authentication Depth** value determines how deeply UniVerse verifies before determining that a certificate is not valid.

Depth is the maximum number of intermediate issue certificates, or CA certificates, UniVerse must examine while verifying an incoming certificate. A depth of 0 indicates that the certificate must be self-signed. A depth of 1 means that the incoming certificate can be either self-signed, or signed by a CA known to the security context.

You can set Authentication Depth on the server and the client. The default value for both is 1.

You can add an authentication rule to a security context. UniVerse uses the



rules during SSL negotiation to determine whether or not to trust the peer. UniVerse supports the following rules:

- **Verification Strength** rule – This rule governs the SSL negotiation and determines whether or not an authentication process is considered successful. There are two levels of security, *generous* and *strict*. If you specify *generous*, the certificate need only contain the subject name (common name) that matches one specified by “PeerName” to be considered valid. If you specify *strict*, the incoming certificate must pass a number of checks, including signature check, expiry check, purpose check, and issuer check.

***Note:** IBM recommends setting the rule to generous only for development or testing purposes.*

- **PeerName** rule – By specifying the **PeerName** rule and attribute mark separated common names in ruleString, trust server/client names will be stored in the context.

During the SSL handshake negotiations, the server sends its certificate to the client. By specifying trust server names, the client can control with which server or servers it should communicate. During the handshake, once the server certificate has been authenticated by way of the issuer (CA) certificate(s). UniVerse compares the subject name contained in the certificate against the trusted server names set in the context. If the server subject name matches one of the trusted names, communication continues, otherwise UniVerse does not establish the connection.

If no trusted peer name is set, any peer is considered legitimate.

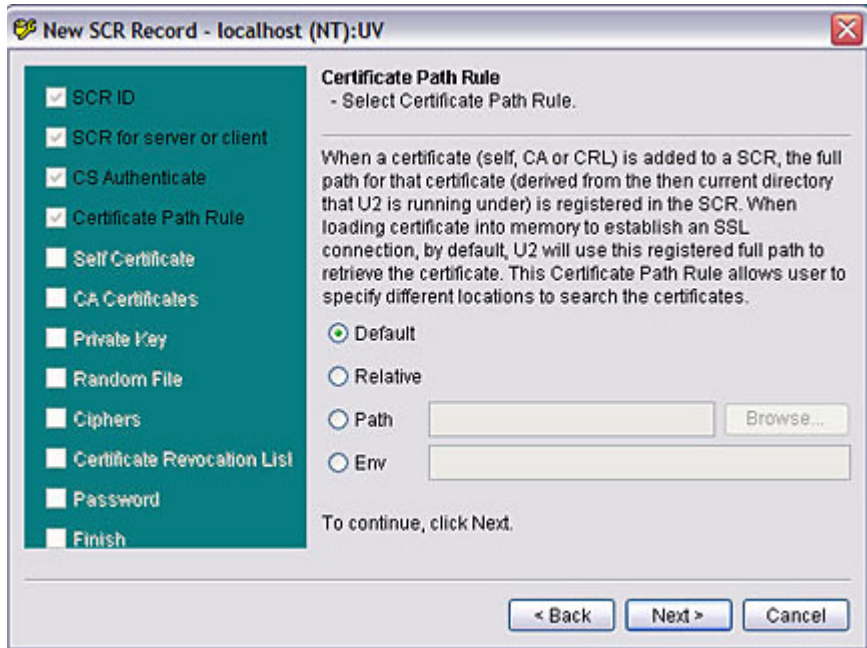
To add an authentication rule, click **Add**. A dialog box similar to the following example appears:



Enter the information for the new rule, then click **OK**.

Select the Authentication Strength, then click **Next**.

6. The **Certificate Path Rule** enables you to specify locations to search for certificates. Define the Certificate Path Rule from the **Certificate Path Rule** dialog box, as shown in the following example:



When you add a certificate to an SCR, the full path for that certificate is registered in the SCR. This path is derived from the current directory where UniVerse is running. When the certificate is loaded into memory to establish the SSL connection, UniVerse uses this registered full path to retrieve the certificate. You can change the path in one of the following ways:

- **Relative** – If you select **Relative**, UniVerse looks in the current directory where UniVerse is running for the certificate.

***Note:** Some of the UniVerse processes, such as the telnet server processes, run from the system directory.*

- **Path** – If you select **Path**, UniVerse uses the path you specify for loading the certificate added to this SCR. You can specify either an absolute path or a relative path.
- **Env** – If you select **Env**, enter an environment variable name in the **Env** box. If you specify this option, the UniVerse process first obtains the value of the environment variable you specify and uses that value as the





path to load the certificates.

Note: UniVerse only evaluates the environment variable when the first SSL connection is made. The value is cached for later reference.

7. You can load a certificate, or multiple certificates, into a security context for use as a UniVerse server certificate or client certificate. Alternatively, you can specify a directory that contains the certificates to use as a Certificate Authority (CA) certificate to authenticate incoming certificates or act as a revocation list, checking for expired or revoked certificates.

The purpose of a certificate is to bind the name of an entity with its public key. It is basically a means of distributing public keys. A certificate always contains the following three pieces of information:

- Name
- Public Key
- Digital signature signed by a trusted third party called a Certificate Authority (CA) with its private key.

If you have the public key of the CA, you can verify that the certificate is authentic.

SSL protocol specifies that when two parties start a handshake, the server must always send its certificate to the client for authentication. It may also require the client to send its certificate to the server for authentication. UniVerse servers that act as HTTP clients are not required to maintain a client certificate. UniVerse applications that act as SSL socket servers must install a server certificate. UniObjects for Java servers and telnet servers also require server certificates.

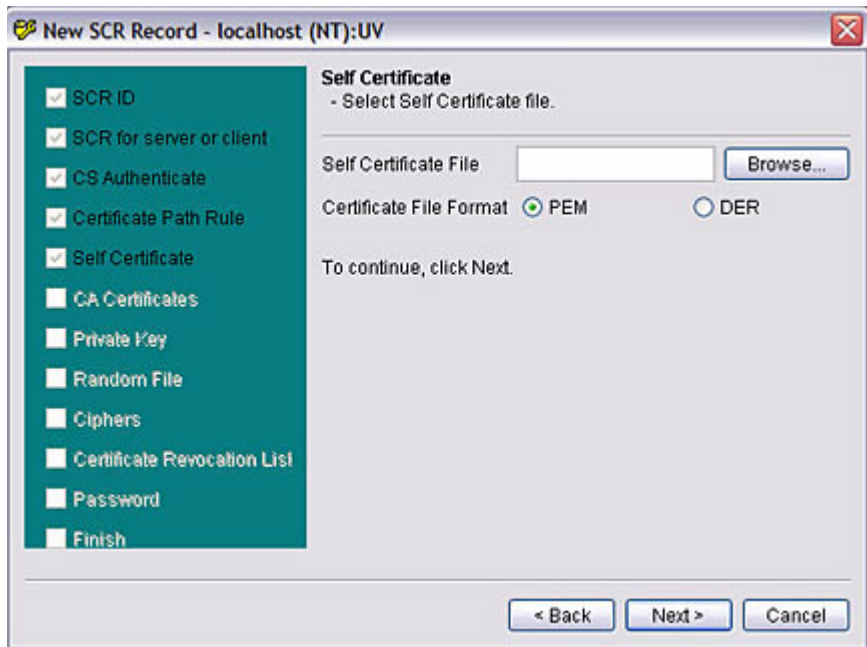
There can be only one server/client certificate per specific security context. Adding a new certificate automatically replaces an existing certificate. However, for issuer certificates, UniVerse chains a new one with existing certificates so UniVerse applications can perform chained authentication.

If the issuer certificate is in PEM format, it can contain multiple certificates generated by concatenating certificates together.

Note: All certificates that form an issuer chain must be of the same type.



Select the self certificate file from the **Self Certificate** dialog box, as shown in the following example:



In the **Self Certificate File** box, enter the path to the file containing the self certificate, or click **Browse** to search for the file.

Next, select the format for the certificate. Select **PEM** for Base64 encoded format, or **DER** for ASN.1 binary format.

Click **Next** to continue.

8. If you are defining an SCR record intended to be used by a server, you must install a private key. If you are defining an SRC record intended to be used by a client and you do not install a self-certificate, you do not need to install a private key.

You can load a private key into a security context so it can be used by SSL functions. Setting a private key replaces an existing private key.

UniVerse uses a private key to digitally sign a message or encrypt a symmetric secret key to use for data encryption.

Select the private key associated with self certificate from the **Private Key** dialog box, as shown in the following example:

New SCR Record - localhost (NT):UV

☒ SCR ID
☒ SCR for server or client
☒ CS Authenticate
☒ Certificate Path Rule
☒ Self Certificate
☒ CA Certificates
☐ Private Key
☐ Random File
☐ Ciphers
☐ Certificate Revocation List
☐ Password
☐ Finish

Private Key File
- Select the private key that corresponds to the self certificate.

Private Key File **Browse...**

Password for Private Key

Private Key Format ☒ PEM ☐ DER

To continue, click Next.

< Back **Next >** **Cancel**

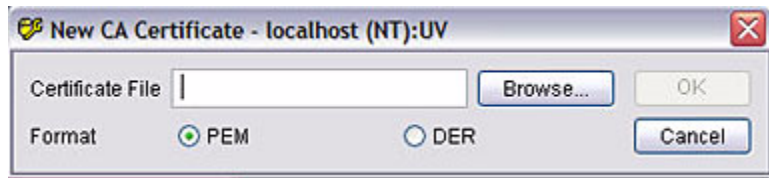
In the **Private Key File** box, enter the file that contains the private key, or click **Browse** to search for the file.

In the **Password for Private Key** box, enter the password for the private key.

Next, select the format for the private key. Click **PEM** for Base64 encoded format, or **DER** for ASN.1 binary format.

Click **Next** to continue.

9. To define a CA certificate, in the **CA Certificate** dialog box, click **Add**. A dialog box similar to the following example appears:



In the **Certificate File** box, enter the path to the file containing the certificate, or click **Browse** to search for the file.

Next, select the format for the certificate. Click **PEM** for Base64 encoded format, or **DER** for ASN.1 binary format.

Click **OK** to add the certificate, or **Cancel** to exit.

Click **Next** to continue.

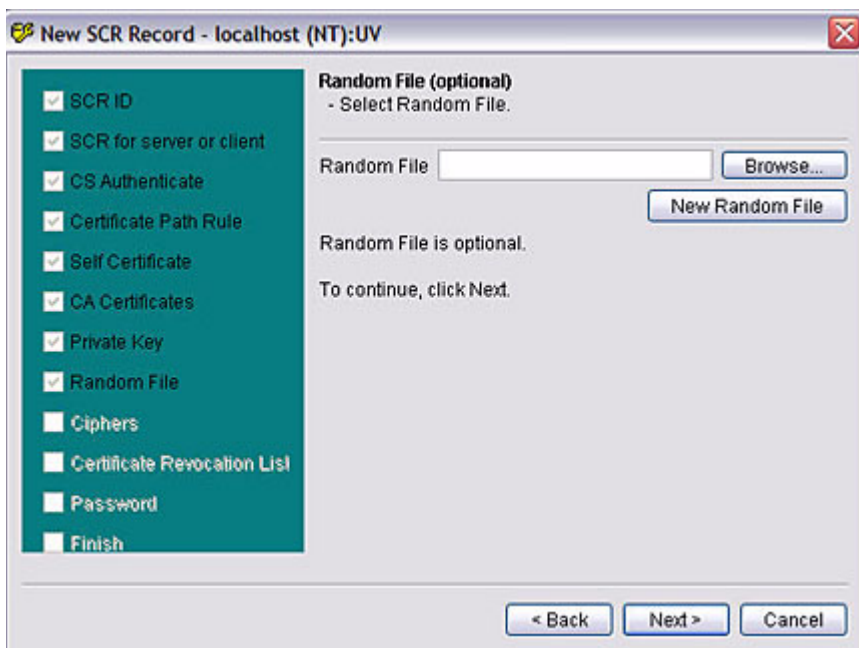


10. You can generate a random file from a series of source files and set that file as the default random file for the SCR record.

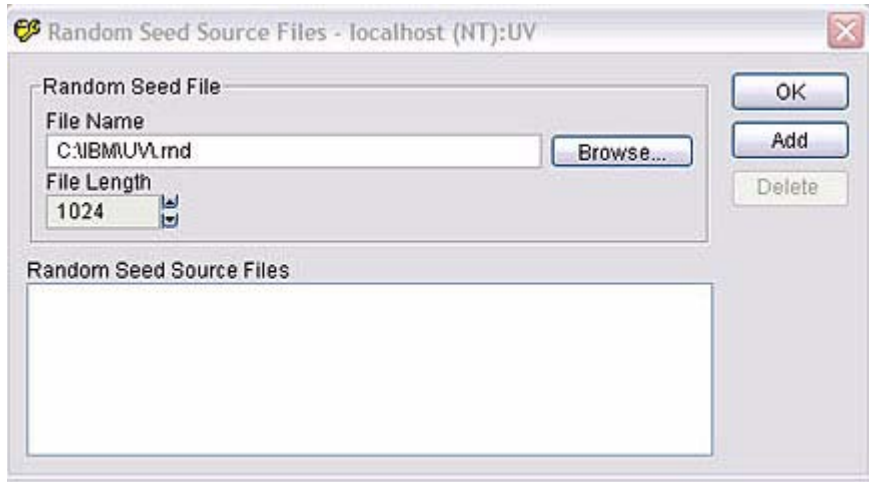
The strength of cryptographic functions depends on the true randomness of the keys. By default, UniVerse uses the .rnd file in the current account. You can override the default by adding a random seed file.

***Note:** IBM recommends you use the default .rnd file.*

To select a random seed file other than the default, click **Browse** to search for the file, as shown in the following example:



To create a new random seed file, click **New Random File**. A dialog box similar to the following example appears:



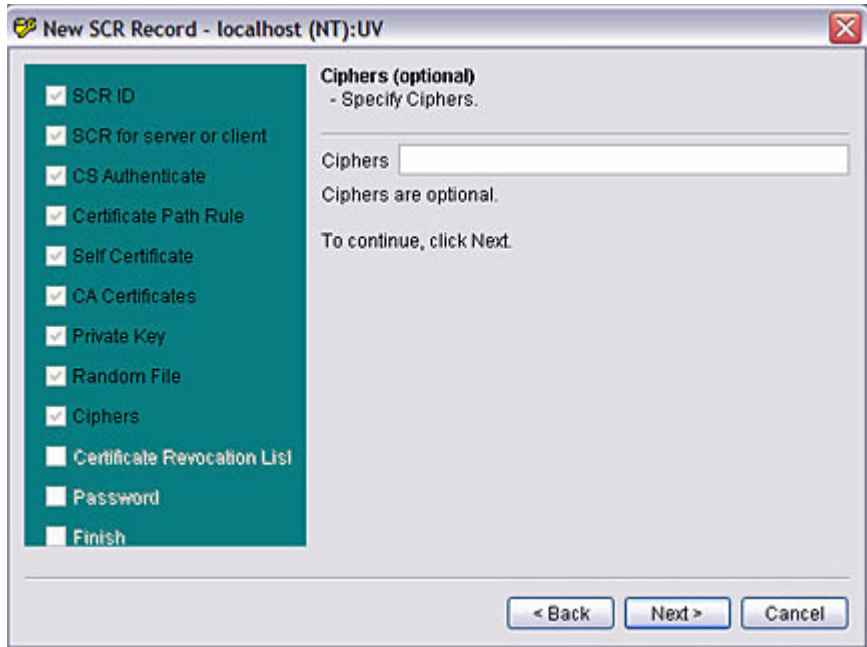
Enter the path to random file, or click **Browse** to select the random source file.

In the **File Length** list, choose the file length.

In the **Random Seed Source Files** box, enter the random seed source files.

Click **OK**.

11. You can define ciphers from the **Ciphers** dialog box, as shown in the following example:



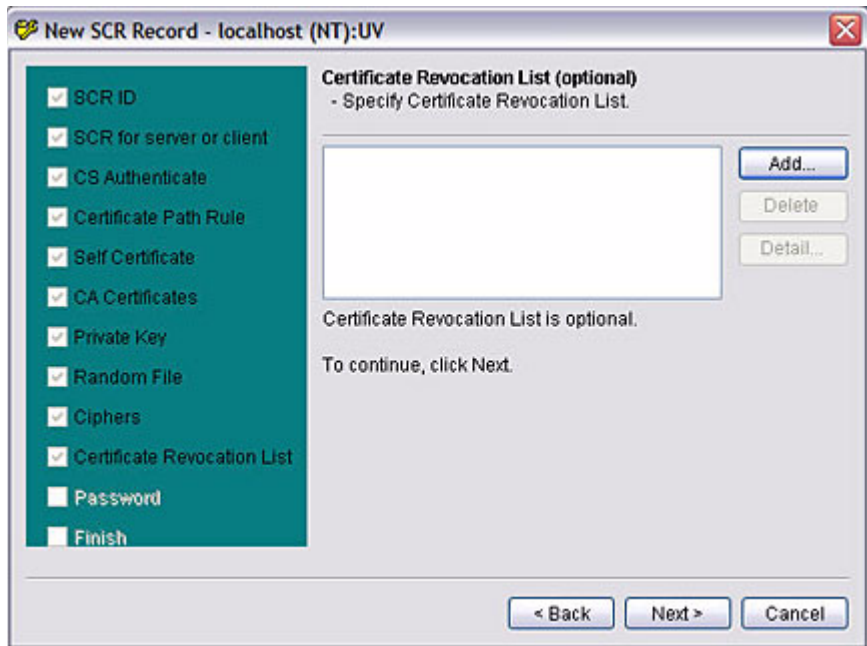
Ciphers enable you to identify which cipher suites should be supported for the specified context. It affects the cipher suites and public key algorithms supported during the SSL/TLS handshake and subsequent data exchanges.

When a context is created, its cipher suites will be set to SSLv3 suites supported by the SSL version you selected.

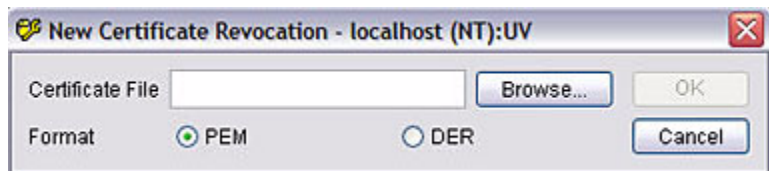
In the **Ciphers** box, enter the Cipher Suite for this SCR record.

The *CipherSpecs* parameter is a string containing *cipher-spec* separated by colons. An SSL cipher specification in *cipher-spec* is composed of 4 major attributes as well as several, less significant attributes. For detailed information about Cipher Suites, see “*UniVerse BASIC Extensions*.”

12. From the **Certificate Revocation List** dialog box, click **Add** to add a CRL file as part of a revocation list, as shown in the following example:



A dialog box similar to the following example appears:



In the **Certificate File** box, enter the path to the file containing the CRL, or click **Browse** to search for the file.

Next, select the format for the CRL file. Click **PEM** for Base64 encoded format, or **DER** for ASN.1 binary format.,

Click **OK** to create the SCR record, or click **Cancel** to exit.

Click **Next** to continue.

13. A dialog box similar to the following example appears:

New SCR Record - localhost (NT):UV

☒ SCR ID
☒ SCR for server or client
☒ CS Authenticate
☒ Certificate Path Rule
☒ Self Certificate
☒ CA Certificates
☒ Private Key
☒ Random File
☒ Ciphers
☒ Certificate Revocation List
☒ Password
☐ Finish

Password
- Type in password for SCR Record.

Password for SCR Record

Confirm Password for SCR Record

To create a new SCR record, click Create.

< Back Create Cancel

In the **Password for SCR Record** box, enter a password to access the record. Reenter the password in the **Confirm Password for SCR Record** box.

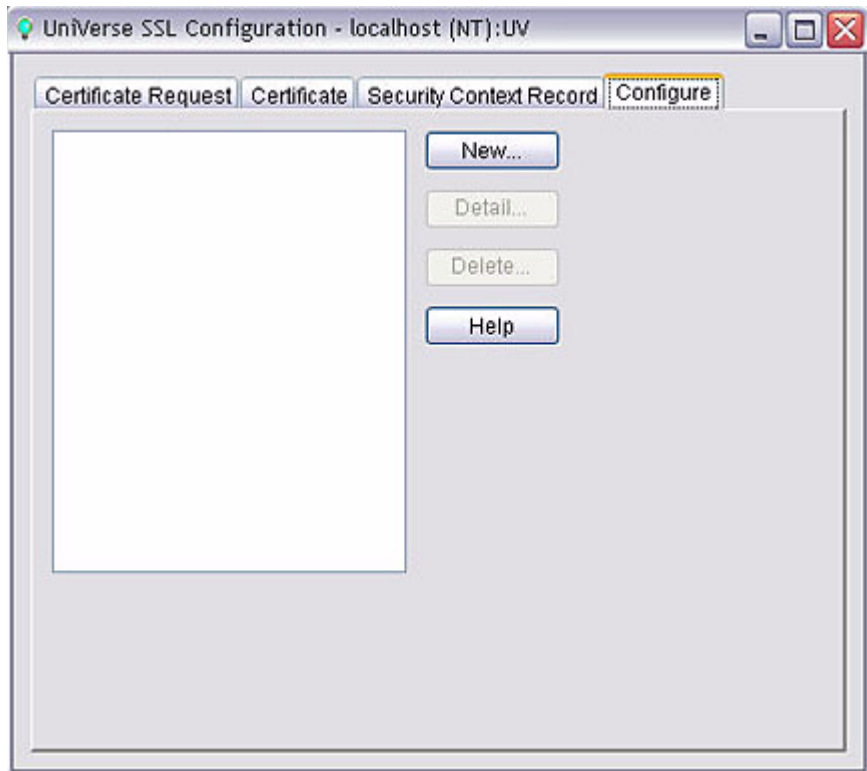
Click **Create** to create the SCR record, or click **Cancel** to exit.

Configuring SSL for UniObjects for Java or Telnet

After you create an SCR record, you need to configure SSL for UniObjects for Java or SSL for Telnet.

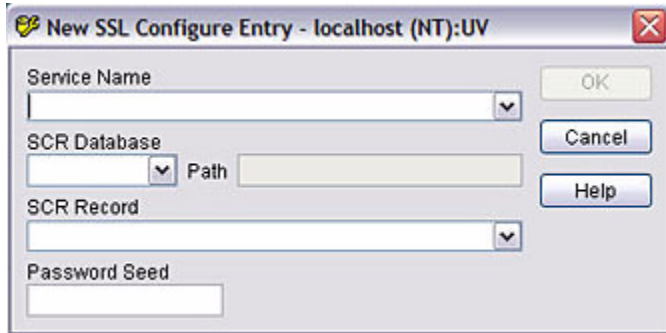
An SCR record contains all SSL-related properties necessary for the server to establish a secured connection with an SSL client. The properties include the server's private key certificate, client authentication flag and strength, and trusted entities. For more information, see “*UniVerse Security Features.*”

From the **UniVerse SSL Configuration** dialog box, select the **Configure** tab. A dialog box similar to the following example appears:



Complete the following steps to configure SSL:

1. Click **New**. A dialog box similar to the following example appears:

A screenshot of a Windows-style dialog box titled "New SSL Configure Entry - localhost (NT):UV". The dialog box has a standard title bar with a yellow icon on the left and a red close button on the right. Inside the dialog, there are four main input sections: "Service Name" with a dropdown menu, "SCR Database" with a dropdown menu and a "Path" text box to its right, "SCR Record" with a dropdown menu, and "Password Seed" with a text box. On the right side of the dialog, there are three buttons: "OK", "Cancel", and "Help".

From the **Service Name** list, select a service name.

From the **SCR Database** list, select the database for this configuration entry. UniAdmin automatically populates the **Path** box.

Select the SCR record from the **SCR Record** list, then enter the password for this record in the **Password Seed** box.

Using SSL with the CallHTTP and Socket Interfaces

Overview of SSL Technology	2-3
Setup and Configuration for SSL.	2-4
SSL Security Programmatic Interfaces for UniData and UniVerse	2-5
Creating A Security Context	2-6
Saving a Security Context	2-8
Loading a Security Context	2-10
Showing a Security Context	2-12
Adding a Certificate	2-13
Adding an Authentication Rule	2-16
Setting a Cipher Suite	2-18
Getting A Cipher Suite	2-26
Setting a Private Key	2-28
Setting Client Authentication Mode	2-31
Setting the Authentication Depth.	2-32
Generating a Key Pair	2-34
Creating a Certificate Request	2-37
Creating a Certificate	2-41
Setting a Random Seed	2-43
Analyzing a Certificate	2-45
Encoding and Cryptographic Functions.	2-46
Encoding Data	2-47
Encrypting Data	2-49
Generating a Message Digest	2-55
Generating a Digital Signature	2-57
Additional Reading	2-60

This chapter describes how to set up and configure SSL for use with the CallHTTP and Socket interfaces.

This chapter consists of the following sections:

- “ Overview of SSL Technology”
- “ Setup and Configuration for SSL”
- “ SSL Security Programmatic Interfaces for UniData and UniVerse”
- “ Encoding and Cryptographic Functions”

Overview of SSL Technology

Secure Sockets Layer (SSL) is a transport layer protocol that provides a secure channel between two communicating programs over which arbitrary application data can be sent securely. It is by far the most widely deployed security protocol used on the World Wide Web.

Although it is most widely used in applications to secure web traffic, SSL actually is a general protocol suitable for securing a wide variety of other network traffic that is based on TCP, such as FTP and Telnet.

SSL provides server authentication, encryption and message integrity. It optionally also supports client authentication.

UniData and UniVerse currently support HTTP and sockets API. SSL support is important for both protocols in order to deploy commercial applications to be able to securely process sensitive data, such as credit card transactions.

Throughout this chapter we talk about SSL exclusively, but in fact we support the more recent development of TLS (Transport Layer Security) protocol, which basically is the adoption of SSL by the standard body IETF and contains support for more public key algorithm and cipher suites.

This document assumes that users who want to use this facility have some basic knowledge of public key cryptography.

Setup and Configuration for SSL

There are no special setup or installation requirements for SSL. The standard installation of the database includes SSL.

SSL Security Programmatic Interfaces for UniData and UniVerse

This section provides information on the SSL functions and properties for UniData and UniVerse.

Many of the functions described in this chapter require as input a pass phrase for various operations. For example, encrypting a generated private key and saving a security context. To ensure a higher level of security, these functions require that pass phrase is assigned a value. General guidelines for passwords should be followed. Particularly, since english text usually has a very low entropy, that is, given part of a word or phrase, the rest isn't completely unpredictable. Thus, it is recommended that the user choose a relatively long phrase, instead of a single word when calling these functions.

Creating A Security Context

The **createSecurityContext()** function creates a security context and returns a handle to the context.

A security context is a data structure that holds all aspects of security characteristics that the application intends to associate with a secured connection. Specifically, the following information may be held for each context:

- Protocol version
- Sender's certificate to be sent to the peer
- Issuer's certificate or certificate chain to be used to authenticate incoming certificate
- Certificate verification depth
- Certificate Revocation List
- Sender's private key for signature and key exchange
- Flag to perform client authentication (useful for server socket only)
- Context ID and time stamp

Syntax

createSecurityContext(*context, version*)

For any given connection, not all of the information is required.

A version (SSL version 2 or 3 or TLS version 1) can be associated with a security context. If no version is provided (i.e. a null string is sent), the default value will be SSL version 3.

For secure socket connections, both socket APIs, **openSecureSocket()** and **initSecureServerSocket()** must be called to associate a security context with a connection.

For secure HTTP connection (https), a valid context handle must be supplied with the **createSecureRequest()** function.

All aspects of a context can be changed by the API's described below.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security context handle.
<i>version</i>	A string with the following values: SSLv2, SSLv3 or TLSv1.

createSecurityContext Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Security context could not be created.
2	Invalid version.

Return Code Status

Saving a Security Context

The **saveSecurityContext()** function encrypts and saves a security context to a system security file. The file is maintained on a per account basis for UniData and UniVerse. The name is used as the record ID to access the saved security information. Since the information is encrypted, the user should not attempt to directly manipulate it.

A user may want his application to a security context to be used later. Multiple contexts may be created to suit different needs. For example, the user may want to use different protocols to talk to different servers. These contexts can be saved and reused.

When creating a saved context, the user must provide both a *name* and a *passPhrase* to be used to encrypt the contents of the context. The *name* and *passPhrase* must be provided to load the saved context back. To ensure a high level of security, it is recommended that the *passPhrase* be relatively long, yet easy to remember.

Syntax

saveSecurityContext(*context, name, passPhrase*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security context handle.
<i>name</i>	String containing the file name of the saved context.
<i>passPhrase</i>	String containing the password to encrypt the context contents.

saveSecurityContext Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid security context handle.
2	Invalid parameters (empty name or passPhrase).
3	Context could not be saved.

Return Code Status

Loading a Security Context

The **loadSecurityContext()** function loads a saved security context record into the current session.

The *name* and *passPhrase* parameters are needed to retrieve and decrypt the saved context. An internal data structure will be created and its handle is returned in the *context* parameter.

Syntax

loadSecurityContext(*context*, *name*, *passPhrase*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The handle to be returned.
<i>name</i>	String containing the name of the file storing the security contents.
<i>PassPhrase</i>	String containing the <i>passPhrase</i> needed to decrypt the saved data.

loadSecurityContext Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Context record does not exist.

Return Code Status

Return Code	Status
2	Context record could not be accessed (e.g. wrong password).
3	Invalid content (file was not saved by the saveSecurityContext() function).
4	Other problems that caused context load failure. Refer to the log file for more information.
Return Code Status (Continued)	



Showing a Security Context

The **showSecurityContext()** function dumps the SSL configuration parameters of a security context into a readable format.

The security context handle must have been returned by a successful execution of **createSecurityContext()** or **loadSecurityContext()**.

The configuration information includes: *protocol*, *version*, *certificate*, *cipher suite* used by this connection and *start time*, etc.

Warning: For security reasons, the *privateKey* installed into the context is not displayed. Once installed, there is no way for the user to extract it.

Syntax

showSecurityContext(*context*,*config*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>config</i>	A dynamic array containing the configuration data.

saveSecurityContext Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Configuration data could not be obtained.

Return Code Status

Adding a Certificate

The **addCertificate()** function loads a certificate (or multiple certificates) into a security context to be used as a UniData or UniVerse server or client certificate. Alternatively, it can specify a directory which contains the certificates that are either used as CA (Certificate Authority) certificates to authenticate incoming certificates or act as a Revocation list to check against expired or revoked certificates.

A certificate's purpose is to bind an entity's name with its public key. It is basically a means of distributing public keys. A certificate always contains three pieces of information: a name, a public key, and a digital signature signed by a trusted third party called a Certificate Authority (CA) with its private key. If you have the CA's public key, you can verify that the certificate is authentic. That is, whether or not the public key contained in the certificate is indeed associated with the entity specified with the name in the certificate. In practice, a certificate can and often does contain more information, for example, the period of time the certificate is valid.

SSL protocol specifies that when two parties start a handshake, the server must always send its certificate to the client for authentication. It may optionally require the client to send its certificate to the server for authentication as well.

Therefore, UniData and UniVerse applications that act as HTTPS clients are not required to maintain a client certificate. The application should work with web servers that do not require client authentication. While UniData and UniVerse applications that do act as SSL socket servers must install a server certificate.

Regardless of which role the application is going to assume, it needs to install a CA certificate or a CA certificate chain to be able to authenticate an incoming certificate.

All certificates are stored in OS level files. Currently, the certificates supported are in conformance with X.509 standards and should be in either DER (Distinguished Encoding Rules, a special case of Abstract Syntax Notation 1, ASN.1) format, or PEM (Privacy Enhanced Mail, an IETF standard) format.

There can be only one server/client certificate per specific security context thus, adding a new certificate will automatically replace an existing certificate. For issuer certificates however, a new one will be chained with existing certificates so UniData and UniVerse applications can perform chained authentication. The new certificate will be added to the end of the chain, meaning that it will be used as the issuer certificate to authenticate the one before it. If the issuer certificate file is in PEM format, it can contain multiple certificates generated by simply concatenating certificates together. The order in which the certificates are stored does make a difference. Note that all certificates that form an issuer chain must be of the same type. That is, they must be either all RSA type or all DSA type. However, you can add both an RSA type and DSA type certificate to a context as specified by the *algorithm* parameter.

If the *certPath* parameter is a directory then all certificates under the directory will be used as issuer certificates when authenticating an incoming certificate.

Syntax

addCertificate(*certPath*, *usedAs*, *format*, *algorithm*, *context*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>certPath</i>	A String containing the name of the OS level file that holds the certificate, or the directory containing certificates.
<i>usedAs</i>	Flag - 1: Used as a Client/Server certificate 2: Used as an issuer certificate 3: Used as a Certificate Revocation List (CRL)
<i>format</i>	Flag - 1: PEM format 2: DER format
<i>algorithm</i>	Flag - 1: RSA key 2: DSA key
<i>context</i>	The Security context handle.

addCertificate Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Certificate file could not be opened or directory does not exist.
3	Unrecognized format.
4	Corrupted or unrecognized certificate contents.
5	Invalid parameter value(s).

Return Code Status

Adding an Authentication Rule

The **addAuthenticationRule()** function adds an authentication rule to a security context. The rules are used during SSL negotiation to determine whether or not the peer is to be trusted.

Currently, the following rules are supported:

Verification Strength rule - This rule governs the SSL negotiation and determines whether or not an authentication process is considered successful. There are two levels of security, *generous* and *strict*. If *generous* is specified, the certificate need only contain the subject name (common name) that matches one specified by “PeerName”, to be considered valid. If *strict* is specified, the incoming certificate must pass a number of checks, including signature check, expiry check, purpose check and issuer check.

***Note:** Setting the rule to **generous** is recommended only for development or testing purposes.*

PeerName rule - By specifying the **PeerName** rule and attribute mark separated common names in ruleString, trusted server/client names will be stored into the context.

During the SSL handshake negotiation, the server will send its certificate to the client. By specifying trusted server names, the client can control which server or servers it should communicate with. During the handshake, once the server certificate has been authenticated by way of the issuer (CA) certificate(s), the subject name contained in the certificate will be compared against the trusted server names set in the context. If the server subject name matches one of the trusted names, communication will continue, otherwise the connection will not be established.

If no trusted peername is set, then any peer is considered legitimate.

Syntax

addAuthenticationRule(context,serverOrClient, rule, ruleString)



Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>ServerOr-Client</i>	Flag 1 - Server Flag 2 -Client Any other value is treated as a value of 1.
<i>Rule</i>	The rule name string. Valid settings are <i>PeerName</i> or <i>VerificationStrength</i> .
<i>RuleString</i>	Rule content string. May be attribute mark separated.

addAuthenticationRule Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Invalid rule name.
3	Invalid rule content.

Return Code Status

Setting a Cipher Suite

The **setCipherSuite()** function allows you to identify which cipher suites should be supported for the specified context. It affects the cipher suites and public key algorithms supported during the SSL/TLS handshake and subsequent data exchanges.

When a context is created, its cipher suites will all be set to SSLv3 suites by default.

The *CipherSpecs* parameter is a string containing *cipher-spec* separated by colons. An SSL cipher specification in *cipher-spec* is composed of 4 major attributes as well as several, less significant attributes. These are defined below.

Some of this information on ciphers is excerpted from the mod_ssl open source package of the Apache web server.

- **Key Exchange Algorithm** - RSA or Diffie-Hellman variants.
- **Authentication Algorithm** - RSA, Diffie-Hellman, DSS or none.
- **Cipher/Encryption Algorithm** - DES, Triple-DES, RC4, RC2, IDEA or none.
- **MAC Digest Algorithm** - MD5, SHA or SHA1.

An SSL cipher can also be an export cipher and is either an SSLv2 or SSLv3/TLSv1 cipher (here TLSv1 is equivalent to SSLv3). To specify which ciphers to use, one can either specify all the ciphers, one at a time, or use aliases to specify the preference and order for the ciphers.

The following table describes each tag for the **Key Exchange Algorithm**.

Tag	Description
<i>KRSA</i>	RSA key exchange
<i>kDHR</i>	Diffie-Hellman key exchange with RSA key
<i>kDHd</i>	Diffie-Hellman key exchange with DSA key
<i>kEDH</i>	Ephemeral (temp.key) Diffie-Hellman key exchange (no cert)

Key Exchange Algorithm Cipher Tags

The following table describes each tag for the **Authentication Algorithm**.

Tag	Description
<i>aNULL</i>	No authentication
<i>aRSA</i>	RSA authentication
<i>aDSS</i>	DSS authentication
<i>aDH</i>	Diffie-Hellman authentication
Authentication Algorithm Cipher Tags	

The following table describes each tag for the **Cipher Encoding Algorithm**.

Tag	Description
<i>eNULL</i>	No encoding
<i>DES</i>	DES encoding
<i>3DES</i>	Triple-DES encoding
<i>RC4</i>	RC4 encoding
<i>RC2</i>	RC2 encoding
<i>IDEA</i>	IDEA encoding
Cipher Encoding Algorithm Cipher Tags	

The following table describes each tag for the **MAC Digest Algorithm**.

Tag	Description
<i>MD5</i>	MD5 hash function
<i>SHA1</i>	SHA1 hash function
<i>SHA</i>	SHA hash function
MAC Digest Algorithm Cipher Tags	

The following table describes each of the **Aliases**.

Alias	Description
<i>SSLv2</i>	all SSL version 2.0 ciphers
<i>SSLv3</i>	all SSL version 3.0 ciphers
<i>TLSv1</i>	all TLS version 1.0 ciphers
<i>EXP</i>	all export ciphers
<i>EXPORT40</i>	all 40-bit export ciphers only
<i>EXPORT56</i>	all 56-bit export ciphers only
<i>LOW</i>	all low strength ciphers (no export, single DES)
<i>MEDIUM</i>	all ciphers with 128 bit encryption
<i>HIGH</i>	all ciphers using Triple-DES
<i>RSA</i>	all ciphers using RSA key exchange
<i>DH</i>	all ciphers using Diffie-Hellman key exchange
<i>EDH</i>	all ciphers using Ephemeral Diffie-Hellman key exchange
<i>ADH</i>	all ciphers using Anonymous Diffie-Hellman key exchange
<i>DSS</i>	all ciphers using DSS authentication
<i>NULL</i>	all cipher using no encryption

Alias Cipher Tags

Now where this becomes interesting is that these can be put together to specify the order and ciphers you wish to use. To speed this up there are also aliases (*SSLv2*, *SSLv3*, *TLSv1*, *EXP*, *LOW*, *MEDIUM*, *HIGH*) for certain groups of ciphers. These tags can be joined together with prefixes to form the *cipher-spec*.

The following table describes the available prefixes.

Tag	Description
<i>none</i>	Add cipher to the list.
+	Add ciphers to the list and pull them to the current location in the list.
-	Remove the cipher from the list (it can be added again later).
!	Kill the cipher from the list completely (cannot be added again later).

Available Prefixes

A more practical way of looking at all of this is to use the **getCipherSuite()** function which provides a nice way to successively create the correct *cipher-spec* string. The default setup for a *cipher-spec* string is shown in the following example:

```
"ALL:!ADH=RC4+RSA:+HIGH:+MEDIUM:+LOW:SSLV2:+EXP"
```

As is shown in the example, you must first remove from consideration any ciphers that do not authenticate, i.e. for SSL only the Anonymous Diffie-Hellman ciphers. Next, use ciphers using RC4 and RSA. Next include the high, medium and then the low security ciphers. Finally pull all SSLv2 and export the ciphers to the end of the list.

The complete list of particular RSA ciphers for SSL is given in the following table.

Cipher Tag	Protocol	Key Ex.	Auth.	Enc.	MAC	Type
<i>DES-CBC3-SHA</i>	SSLv3	RSA	RSA	3DES(168)	SHA1	
<i>DES-CBC3-MD5</i>	SSLv2	RSA	RSA	3DES(168)	MD5	
<i>IDEA-CBC-SHA</i>	SSLv3	RSA	RSA	IDEA(128)	SHA1	
<i>RC4-SHA</i>	SSLv3	RSA	RSA	RC4(128)	MD5	
<i>RC4-MD5</i>	SSLv3	RSA	RSA	RC4(128)	MD5	
<i>IDEA-CBC-MD5</i>	SSLv2	RSA	RSA	IDEA(128)	MD5	
<i>RC2-CBC-MD5</i>	SSLv2	RSA	RSA	RC2(128)	MD5	
<i>RC4-MD5</i>	SSLv2	RSA	RSA	RC4(128)	MD5	
<i>DES-CBC-SHA</i>	SSLv3	RSA	RSA	DES(56)	SHA1	
<i>RC4-64-MD5</i>	SSLv2	RSA	RSA	RC4(64)	MD5	
<i>DES-CBC-MD5</i>	SSLv2	RSA	RSA	DES(56)	MD5	
<i>EXP-DES-CBC-SHA</i>	SSLv3	RSA(5 12)	RSA	DES(40)	SHA1	export
<i>EXP-RC2-CBC-MD5</i>	SSLv3	RSA(5 12)	RSA	RC2(40)	MD5	export
<i>EXP-RC4-MD5</i>	SSLv3	RSA(5 12)	RSA	RC4(40)	MD5	export
<i>EXP-RC2-CBC-MD5</i>	SSLv2	RSA(5 12)	RSA	RC2(40)	MD5	export
<i>EXP-RC4-MD5</i>	SSLv2	RSA(5 12)	RSA	RC4(40)	MD5	export
<i>NULL-SHA</i>	SSLv3	RSA	RSA	None	SHA1	
<i>NULL-MD5</i>	SSLv3	RSA	RSA	None	MD5	
RSA Ciphers						

The complete list of particular DH ciphers for SSL is given in the following table.

Cipher Tag	Protocol	Key Ex.	Auth.	Enc.	MAC	Type
<i>ADH-DES-CBC3-SHA</i>	SSLv3	DH	None	3DES(168)	SHA1	
<i>ADH-DES-CBC-SHA</i>	SSLv3	DH	None	DES(56)	SHA1	
<i>ADH-RC4-MD5</i>	SSLv3	DH	None	RC4(128)	MD5	
<i>EDH-RSA-DES-CBC3-SHA</i>	SSLv3	DH	RSA	3DES(168)	SHA1	
<i>EDH-DSS-DES-CBC3-SHA</i>	SSLv3	DH	DSS	3DES(168)	SHA1	
<i>EDH-RSA-DES-CBC-SHA</i>	SSLv3	DH	RSA	DES(56)	SHA1	
<i>EDH-DSS-DES-CBC-SHA</i>	SSLv3	DH	DSS	DES(56)	SHA1	
<i>EXP-EDH-RSA-DES-CBC-SHA</i>	SSLv3	DH(512)	RSA	DES(40)	SHA1	export
<i>EXP-EDH-DSS-DES-CBC-SHA</i>	SSLv3	DH(512)	DSS	DES(40)	SHA1	export
<i>EXP-ADH-DES-CBC-SHA</i>	SSLv3	DH(512)	None	DES(40)	SHA1	export
<i>EXP-ADH-RC4-MD5</i>	SSLv3	DH(512)	None	RC4(40)	MD5	export

Diffie-Hellman Ciphers

Example:

```
SetCipherSuite(ctxHandle, "RSA:!EXP:!NULL:+HIGH:+MEDIUM:-LOW")
SetCipherSuite(ctxHandle, "SSLv3")
```

Syntax

setCipherSuite(context, cipherSpecs)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>CipherSpecs</i>	String containing cipher suite specification described above.

setCipherSuite Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Invalid cipher suite specification.

Return Code Status

Getting A Cipher Suite

The **getCipherSuite()** function obtains information about supported cipher suites, their version, usage, strength and type for the specified security *context*. The result is put into the dynamic array *ciphers*, with one line for each cipher suite, separated by a field mark (@FM). The format of the string for one cipher suite is as follows.

Suite, version, key-exchange, authentication, encryption, digest, export

Refer to the cipher tables under the “[Setting a Cipher Suite](#),” section for definitions of all suites. The following is an example of a typical Suite.

```
EXP-DES-CBC-SHA SSLv3 Kx=RSA(512) Au=RSA Enc=DES(40) Mac=SHA1
export
```

The suite is broken down as follows. The suite name is EXP-DES-CBC-SHA. It is specified by SSLv3. The Key-exchange algorithm is RSA with 512-bit key. The authentication is also done by RSA algorithm. The Data encryption uses DES (Data Encryption Standard, an NIST standard) with CBC mode. MAC (Message Authentication Code, a hash method to calculate message digest) will be done with SHA-1 (Secure Hash Algorithm 1, also an NIST standard) algorithm. The suite is exportable.

Only those methods that are active for the protocol will be retrieved.

Syntax

```
getCipherSuite(context,ciphers)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>ciphers</i>	A Dynamic array containing the cipher strings delimited by @FM.

getCipherSuite Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Unable to obtain information.

Return Code Status

Setting a Private Key

The **setPrivateKey()** function loads the private key into a security context so that it can be used by SSL functions. If the context already had a set private key, it will be replaced.

SSL depends on public key crypto algorithms to perform its functions. A pair of keys is needed for each communicating party to transfer data over SSL. The public key is usually contained in a certificate, signed by a CA, while the private key is kept secretly by the user.

Private key is used to digitally sign a message or encrypt a symmetric secret key to be used for data encryption.

The *Key* parameter contains either the key string itself or a path that specifies a file that contains the key. UniData and UniVerse only accept PKCS #8 style private key.

The *Format* parameter specifies if the key is in binary format or Base64 encoded format. If the key is in a file, Base64 format also means that it must be in PEM format.

The *KeyLoc* parameter specifies if the key is provided in a file or in a dynamic array string.

If the key is previously encrypted, a correct *passPhrase* must be given to decrypt the key first. It is recommended that the private key be always in encrypted form. Note that if the private key is generated by the **generateKey()** function described under the “[Generating a Key Pair](#)” section, then it is always in PEM format and always encrypted by a pass phrase.

If the *validate* parameter is set, then the private key is verified with the public key contained in the certificate specified for either the server or client. They must match for SSL to work. In some cases there is no need or it is impossible to check against a certificate. For example, the certificate is already distributed to the other end and there is no need for user application to authenticate itself. In that case, *validate* can be set to 0.

If *validate* is required, the corresponding certificate should be added first by calling the **addCertificate()** function which is described under the “[Adding a Certificate](#)” section.

The direct form of this function may be preferred by some applications where a hard coded private key can be incorporated into the application, eliminating the need to access an external key file, which may be considered a security hazard.



Private key is the single most important piece of information for a crypto system. You must take every precaution to keep it secure. If the private key is compromised, there will be no data security. This is especially true for server private keys.

Syntax

setPrivateKey(key, format, keyLoc, passPhrase, validate, context)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>Key</i>	A string containing either the key or path for a key file.
<i>Format</i>	1 - PEM (Base64 encoded) format 2 - DER (ASN.1 binary) format
<i>KeyLoc</i>	1 - key contained in key string 2 - key is in a file specified by key
<i>passPhrase</i>	String containing the path phrase required for gaining access to the key. It can be empty if the key is not pass phrase protected. THIS IS NOT RECOMMENDED!
<i>Validate</i>	1 - Validate against matching public key 0 - Won't bother to validate
<i>Context</i>	The security context handle.

setPrivateKey Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success
1	Invalid Security handle
2	Invalid format

Return Code Status

Return Code	Status
3	Invalid key type
4	Key file cannot be accessed (non-existent or wrong pass phrase)
5	Certificate cannot be accessed
6	Private key does not match public key in certificate
7	Private key can't be interpreted
99	Other errors that prevent private key from being accepted by UniData or UniVerse.

Return Code Status (Continued)

Setting Client Authentication Mode

The **setClientAuthentication()** function turns client authentication for a server socket on or off.

When *option* is set to on, during the initial SSL handshake, the server will send client authentication request to the client. It will also receive the client certificate and perform authentication according to the issuer's certificate (or certificate chain) set in the security context.

Syntax

setClientAuthentication(*context*,*option*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>option</i>	1 - ON 2 - OFF

setClientAuthentication Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.

Return Code Status

Setting the Authentication Depth

The **setAuthenticationDepth()** function sets how deeply UniData and UniVerse should verify before deciding that a certificate is not valid.

This function can be used to set both server authentication and client certification, determined by the value in parameter *serverOrClient*. The default depth for both is 1.

The *depth* is the maximum number of intermediate issuer certificate, or CA certificates which must be examined while verifying an incoming certificate. Specifically, a depth of 0 means that the certificate must be self-signed. A default depth of 1 means that the incoming certificate can be either self-signed, or signed by a CA which is known to the *context*.

Syntax

setAuthenticationDepth(*context*, *depth*, *serverOrClient*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>context</i>	The Security Context handle.
<i>depth</i>	Numeric value for verification depth.
<i>serverOr-Client</i>	1 - Server 2 - Client

setAuthenticationDepth Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid Security Context handle.
2	Invalid depth (must be greater than or equal to 0).
3	Invalid value for serverOrClient (must be 1 or 2)

Return Code Status

Generating a Key Pair

The **generateKey()** function generates a public key cryptography key pair and encrypts the private key. You should then put it into an external key file protected by the provided pass phrase. The protected private key can later be used by UniData and UniVerse SSL sessions (via **setPrivateKey()**) to secure communication. The public key will not be encrypted.

The generated private key will be in PKCS #8 form and is encoded in either PEM or DER format specified by *format*. The generated public key is in traditional form. If *keyLoc* is 1, the resulted key is put into a dynamic array in *privKey* and *pubKey*. Otherwise they are put into OS level files specified by *privKey* and *pubKey*.

This function can generate two types of keys, RSA and DSA, specified by *algorithm*. The key length is determined by *keyLength* and must be in the range of 512 to 2048.

For DSA key generation, *paramFile* must be specified. If a parameter file is provided through *paramFile* and it contains valid parameters, then the parameters are used to generate a new key pair. If the specified file does not exist or does not contain valid parameters, a new group of parameters will be generated and subsequently used to generate a DSA key pair. The generated parameters are then written to the specified parameter file. Since DSA parameter generation is time consuming, it is recommended that a parameter file be used to generate multiple DSA key pairs.

To make sure the private key is protected, a pass phrase **MUST** be provided. A one-way hash function will be used to derive a symmetric key from the pass phrase to encrypt the generated key. When installing the private key into a security context with the **setPrivateKey()** function, or generating a certificate request with the **generateCertRequest()** function, this pass phrase must be supplied to gain access to the private key.

Syntax

```
generateKey(privKey, pubKey, format, keyLoc, algorithm, keyLength,  
passPhrase, paramFile)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>privKey</i>	A string storing the generated private key or name of the file storing the generated private key.
<i>pubKey</i>	A string storing the generated public key or name of the file storing the generated public key.
<i>format</i>	1 - PEM 2 - DER
<i>keyLoc</i>	1 - Put the key into string <i>privKey/pubKey</i> . 2 - Put the key into a file.
<i>algorithm</i>	1 - RSA 2 - DSA
<i>keyLength</i>	Number of bits for the generated key. Between 512 and 2048.
<i>passPhrase</i>	A string storing the pass phrase to protect the private key.
<i>paramFile</i>	A parameter file needed by DSA key generation.

generateKey Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Key pair cannot be generated.
2	Unrecognized key file format.
3	Unrecognized encryption algorithm.
4	Unrecognized key type or invalid key length (must be between 512 and 2048).
5	Empty pass phrase.

Return Code Status

Return Code	Status
6	Invalid DSA parameter file.
7	Random number generator cannot be seeded properly.
8	Private key cannot be written.
Return Code Status (Continued)	

Creating a Certificate Request

The **createCertRequest()** function generates a PKCS #10 certificate request from a private key in PKCS #8 form and a set of user specified data. The request can be sent to a CA or used as a parameter to **createCertificate()** as described in “[Creating a Certificate](#)” to obtain an X.509 public key certificate.

The private key and its format, type, algorithm and pass phrase are specified the same as described in the “[Generating a Key Pair](#),” section above.

The certificate request will typically contain the information described in the following table.

Item	Description
<i>Version</i>	Defaults to 0.
<i>Subject</i>	The certificate holder’s identification data. This includes, country, state/province, locality (city), organization, unit, common name, email address, etc.
<i>Public key</i>	The key’s algorithm (RSA or DSA) and value.
<i>Signature</i>	The requester’s signature, (signed by the private key).

Certificate Request Information

The subject data must be provided by the requester through the dynamic array, *subjectData*. It contains @FM separated attributes in the form of “attri=value”.

The commonly used *subjectData* attributes are described in the following table.

Item	Description	Example
<i>C</i>	Country	C=US
<i>ST</i>	State	ST=Colorado
<i>L</i>	Locality	L=Denver

subjectData Attributes



Item	Description	Example
<i>O</i>	Organization	O=MyCompany
<i>OU</i>	Organization Unit	OU=Sales
<i>CN</i>	Common Name	CN=service@mycompany.com
<i>Email</i>	Email Address	Email=john.doe@mycompany.com

subjectData Attributes

Be aware that since the purpose of a certificate is to associate the certificate's bearer with his or her identity, in order for the outside party to verify the identity of the certificate's holder, some recognizable characteristics should be built between the holder and verifier. For example, it is a general practice that a server's certificate uses its domain name (such as myServer.com) as its common name (CN).

Digest specifies what algorithm is going to be used to generate a Message Authentication Code (MAC) which will then be signed with the provided private key as a digital signature as part of the request. Currently only two algorithms, MD5 and SHA1, are supported.

Note: For a DSA request, SHA1 will always be used.

For more information on certificates, see the documentation for X.509 and PKCS #10.

Syntax

createCertRequest(*key, inFormat, keyLoc, algorithm, digest, passPhrase, subjectData, outFile, outFormat*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key</i>	A string containing the key or name of the file storing the key.
<i>inFormat</i>	The key format. 1 - PEM 2 - DER
<i>keyLoc</i>	1 - Put the key into string privKey/pubKey. 2 - Put the key into a file.
<i>algorithm</i>	1 - RSA 2 - DSA
<i>digest</i>	1 - MD5 2 - SHA1
<i>passPhrase</i>	A string storing the pass phrase to protect the private key.
<i>subjectData</i>	The Requester's identification information.
<i>outFile</i>	A string containing the path name of the file where the certificate request is stored.
<i>outFormat</i>	The generated certificate format. 1 - PEM 2 - DER

createCertRequest Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Private key file cannot be opened.
2	Unrecognized key or certificate format.
3	Unrecognized key type.

Return Code Status

Return Code	Status
4	Unrecognized encryption algorithm.
5	Unrecognized key (corrupted key or algorithm mismatch).
6	Invalid pass phrase.
7	Invalid subject data (illegal format or unrecognized attribute, etc.).
8	Invalid digest algorithm.
9	Output file cannot be created.
99	Cert Request cannot be generated.

Return Code Status (Continued)

Creating a Certificate

The **createCertificate()** function generates a certificate. The certificate can either be a self-signed certificate as a root CA that can then be used later to sign other certificates, or it can be a CA signed certificate. The generated certificate conforms to X509V3 standard.

As input, a certificate request file must be specified by *req*. Three actions can be chosen: self-signing, CA-signing or leaf-CA-signing. For self-signing, a key file must be specified by *signKey*. For the other two actions, a CA certificate file must be specified by *CAcert*, along with the CA private key specified by *signKey*. The output certificate file is specified by *certOut*. The format for these files should all be in PEM format.

The difference between CA-signing and leaf-CA-signing is that, for CA-signing, the resultant certificate can serve as an intermediate CA certificate to sign other certificates, while leaf-CA-signing generates certificates that are intended for end user use only.

The *days* parameter specifies the number of days the generated certificate is valid. The certificate is valid starting from the current date until the number of days specified expires. If an invalid *days* value is provided (0 or negative) the default value of 365 (one year) will be used.

This function is provided mainly for the purpose of enabling application development and testing. As such, the certificate generated contains only a minimum amount of information and does not allow any extensions specified by the X509 standard and that are supported by many other vendors. It is recommended that you implement a complete PKI solution partnered with a reputed PKI solution vendor.



Syntax

```
createCertificate(action, req, signKey, keyPass, CAcert, days, extensions, certOut)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>action</i>	1 - Self-signing 2 - CA-signing 3 - leaf-CA-signing
<i>req</i>	A string containing the certificate request file name.
<i>signKey</i>	A String containing the private key file name.
<i>keyPass</i>	A string containing the pass phrase to protect the private key.
<i>CAcert</i>	A string containing the CA certificate.
<i>days</i>	The number of days the certificate is valid for. The default is 365 days.
<i>extensions</i>	A string containing extension specifications.
<i>certOut</i>	A string containing the generated certificate file.

createCertificate Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Cannot read certificate request file.
2	Cannot read the key file.
3	Cannot read the CA certificate file.
4	Cannot generate the certificate.

Return Code Status

Setting a Random Seed

The **setRandomSeed()** function generates a random seed file from a series of source files and sets that file as the default seed file for the supplied security context.

The strength of cryptographic functions depends on the true randomness of the keys. This function generates and sets the random seed file used by many of the UniData and UniVerse cryptographic functions. By default, UniData and UniVerse will use the .rnd file in your current UDTHOME or UVHOME directory. You can override the default by calling this function.

The random seed file is specified by *outFile*, which is generated based on source files specified in *inFiles*. For Windows platforms, multiple files must be separated by “;” (a semi-colon). For Unix platforms, multiple files must be separated by “:” (a colon).

The *length* parameter specifies how many bytes of seed data should be generated.

If no source is specified in the *inFiles* parameter, then the *outFile* parameter must already exist.

If context is not specified, the seed file will be used as a global seed file that applies to all cryptographic functions. However, a seed file setting in a particular security context will always override the global setting.

Syntax

setRandomSeed(*inFiles*, *outFile*, *length*, *context*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>inFiles</i>	A string containing source file names.
<i>outFiles</i>	A string containing the generated seed file.
<i>length</i>	The number of bytes that should be generated (the default is 1024 if less than 1024 is specified).
<i>context</i>	The Security Context handle.

setRandomSeed Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid parameter(s).
2	Random file generation error.
3	Random file set error.

Return Code Status

Analyzing a Certificate

The **analyzeCertificate()** function decodes a certificate and inputs plain text into the *result* parameter. The *result* parameter will then contain such information as the subject name, location, institute, issuer, public key, other extensions and the issuer’s signature.

Syntax

```
analyzeCertificate(cert, format, result)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>cert</i>	A string containing the certificate file name.
<i>format</i>	1 - PEM 2 - DER
<i>result</i>	A dynamic array containing parsed cert data, in ASCII format.

analyzeCertificate Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Failed to open cert file.
2	Invalid format.
3	Unrecognized cert.
4	Other errors.

Return Code Status

Encoding and Cryptographic Functions

This section describes the available encoding and cryptographic functions included with this implementation of SSL.

The main purpose of data encoding is to allow the use of non-ASCII characters in a body of data such that the data can be transferred undisturbed by underlying protocols or displayed without causing problems.

Encoding Data

The **ENCODE()** function performs data encoding on input data. Currently only Base64 encoding is supported. Base 64 encoding is designed to represent arbitrary sequences of octets that do not need to be humanly readable. A 65-character subset of US-ASCII is used, enabling 6-bits to be represented per printable character. The subset has the important property that it is represented identically in all versions of ISO646, including US-ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. The encoded output stream must be represented in lines of no more than 76 characters each. All line breaks must be ignored by the decoding process. All other characters not found in the 65-character subset should trigger a warning by the decoding process.

The function can perform either encoding or decoding, as specified by *action*. The data can either be in the dynamic array, *data*, or in a file whose name is specified in *data*, determined by *dataLoc*.

Syntax

ENCODE(*algorithm*, *action*, *data*, *dataLoc*, *result*, *resultLoc*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>algorithm</i>	A string containing the encode method name. Base64 is currently the only supported method.
<i>action</i>	1 - Encode 2 - Decode
<i>data</i>	Data or the name of the file containing the data to be encoded or decoded.

ENCODE Parameters

Parameter	Description
<i>dataLoc</i>	1 - Data in a string 2 - Data in a file
<i>result</i>	Encoded or decoded data or the name of the file storing the processed data.
<i>resultLoc</i>	1 - Result in a string 2 - Result in a file.

ENCODE Parameters (Continued)

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Unsupported algorithm.
2	Invalid parameters (invalid data or result location type, etc.).
3	The data cannot be read.
4	The data cannot be encoded or decoded.

Return Code Status

Encrypting Data

The **ENCRYPT()** function performs symmetric encryption operations. Various block and stream symmetric ciphers can be called through this function. The supported ciphers are listed below.

Ciphers are specified by algorithm and are not case sensitive. Base64 encoding and decoding can be specified with the *action* parameter. If encoding is specified, the encrypted data is Base64 encoded before being entered into *result*. If decoding is specified, the data is Base64 decoded before being encrypted. The data and its location are specified by *data* and *dataLoc*, respectively. *Key* can be explicitly specified or read from a file, or, alternatively, derived on the fly, specified by *keyAction*, in which case the *key* string is used as a pass phrase to derive the actual key. The encrypted or decrypted data is put into the dynamic array *result*, or a file, as specified by *resultLoc*.

Salt is used to provide more security against certain kinds of cryptanalysis attacks, such as dictionary attacks. If an empty *salt* is supplied, an internally generated salt will be used in deriving the key. *Salt* is ignored when *action* is set to decrypt. *IV* (Initialization Vector) is used to provide additional security to some block ciphers. It does not need to be secret but should be fresh, meaning different for each encrypted data. If an existing key is supplied, *IV* is generally needed. However if the encryption key is to be derived from a pass phrase, *IV* can be generated automatically. Both *salt* and *IV* must be provided in hexadecimal format.



Note: Some ciphers are more secure than others. For more details, please refer to the publications listed under “ [Additional Reading](#).”

The following ciphers are supported. All cipher names are not case sensitive.



Note: Due to export restrictions, all ciphers may not be available for a specific distribution.

56-bit key DES algorithms:

Algorithm	Description
des-cbc	DES in CBC mode
des	Alias for des-cbc
des-cfb	DES in CFB mode
des-ofb	DES in OFB mode
des-ecb	DES in ECB mode
56-bit DES algorithms	

112-bit key DES algorithms:

Algorithm	Description
des-edc-cbc	Two key triple DES EDE in CBC mode
des-edc	Alias for des-edc-cbc
des-edc-cfb	Two key triple DES EDE in CFB mode
des-edc-ofb	Two key triple DES EDE in OFB mode
112-bit DES algorithms	

168-bit key DES algorithms:

Algorithm	Description
des-ede3-cbc	Three key triple DES EDE in CBC mode
des-ede3	Alias for des-ede3-cbc
des3	Alias for des-ede3-cbc
des-ede3-cfb	Three key triple DES EDE in CFB mode
des-ede3-ofb	Three key triple DES EDE in OFB mode
168-bit DES algorithms	

RC2 algorithms:

Algorithm	Description
rc2-cbc	128-bit RC2 in CBC mode
rc2	Alias for rc2-cbc
rc2-cfb	128-bit RC2 in CBC mode
rc2-ecb	128-bit RC2 in ECB mode
rc2-ofb	128-bit RC2 in OFB mode
rc2-64-cbc	64-bit RC2 in CBC mode
rc2-40-cbc	40-bit RC2 in CBC mode
RC2 algorithms	

RC4 algorithms:

Algorithm	Description
rc4	128-bit RC4
rc4-64	64-bit RC4
rc4-40	40-bit RC4
RC4 algorithms	

RC5 algorithms (all 128-bit key):

Algorithm	Description
rc5-cbc	RC5 in CBC mode
rc5	Alias for rc5-cbc
RC5 algorithms	

Algorithm	Description
rc5-cfb	RC5 in CFB mode
rc5-ecb	RC5 in ECB mode
rc5-ofb	RC5 in OFB mode
RC5 algorithms	

Syntax

ENCRYPT(*algorithm, action, data, dataLoc, key, keyLoc, keyAction, salt, IV, result, resultLoc*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>algorithm</i>	A string containing the cipher name.
<i>action</i>	1 - Encrypt 2 - Base64 encode after encryption 3 - Decrypt 4 - Base64 decode before encryption
<i>data</i>	Data or the name of the file containing the data to be processed.
<i>dataLoc</i>	1 - Data in a string 2 - Data in a file
<i>key</i>	The actual key (password) or file name containing the key.
<i>keyLoc</i>	1 - Key in data 2 - Key in file
<i>keyAction</i>	1 - Use actual key 2 - Derive key from pass phrase
<i>Salt</i>	A string containing the Salt value.

ENCRYPT Parameters

Parameter	Description
<i>IV</i>	A string containing IV.
<i>result</i>	The result buffer or the name of the file storing the result.
<i>resultLoc</i>	1 - Result in a string 2 - Result in a file.

ENCRYPT Parameters (Continued)

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Invalid cipher.
2	Invalid parameters (location/action value is out of range, etc.).
3	The data cannot be read.
4	The key cannot be derived.
5	Base 64 encoding/decoding error.
6	Encryption/decryption error.

Return Code Status

Generating a Message Digest

The **DIGEST()** function generates a message digest of supplied data. A message digest is the result of a one-way hash function (digest algorithm) performed on the message. Message digest has the unique properties that a slight change in the input will result in a significant difference in the resulting digest. Therefore, the probability of two different messages resulting in the same digest (collision) is very unlikely. It is also virtually impossible to reverse to the original message from a digest. Message digest is widely used for digital signatures and other purposes.

The desired digest algorithm is specified in *algorithm*. The two supported digest algorithms are **MD5** (Message Digest 5, 128-bit) and **SHA1** (Secure Hash Algorithm 1, 160-bit). Data and its location are specified by *data* and *dataLoc*, respectively. The arrived digest will be put into a dynamic array in *result*. Since digest is short and has a fixed length, it is always put into a string and no file option is provided. The result can be in either binary or hex format.

Syntax

DIGEST(*algorithm*, *data*, *dataLoc*, *result*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>algorithm</i>	A string containing the digest algorithm name (either “MD5” or “SHA1”).
<i>data</i>	Data or the name of the file containing the data to be digested.
<i>dataLoc</i>	1 - Data in a string 2 - Data in a file
<i>result</i>	A string to store the digest result.

DIGEST Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Unsupported digest algorithm.
2	The data file cannot be read.
3	Message digest cannot be obtained.
4	Invalid parameters.

Return Code Status

Generating a Digital Signature

The **SIGNATURE()** function generates a digital signature or verifies a signature using the supplied key.

The *algorithm* parameter specifies the digest algorithm used to construct the signature. The supported algorithms are **MD5** and **SHA1**. There are four actions that can be specified: **RSA-Sign**, **RSA-Verify**, **DSA-Sign**, and **DSA-Verify**. Note that if DSA is chosen, only **SHA1** can be specified in *algorithm*.

The data to be signed or verified against a signature can be supplied either directly in *data*, or read from a file whose names is in *data*.

For signing action, a private key should be specified. For verification, a public key is usually expected. However, a private key is also accepted for verification purposes. *Key* can be either in PEM or DER format. If a private key is password protected, the password must be supplied with *pass*.

For verification, *key* can also contain a certificate or name of a certificate file. A signature is expected in *sigIn*.

For signing action, the generated signature is put into *result*.

Syntax

SIGNATURE(*algorithm, action, data, dataLoc, key, keyLoc, keyFmt, pass, sigIn, result*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>algorithm</i>	The digest algorithm used for signing or verification (must be either “MD5” or “SHA1”).
<i>action</i>	1 - RSA-Sign 2 - RSA-Verify 3 - DSA-Sign 4 - DSA-Verify
<i>data</i>	Data or the name of the file containing the data to be signed or verified.
<i>dataLoc</i>	1 - Data in a string 2 - Data in a file
<i>key</i>	The key or the name of the file containing the key to be used to sign or verify. In the case of verification, key can be a certificate string or a file.
<i>keyLoc</i>	1 - Key is in a string 2 - Key is in a file 3 - Key is in a certificate for verification
<i>keyFmt</i>	1 - PEM 2 - DER
<i>pass</i>	A string containing the pass phrase for the private key.
<i>sigIn</i>	A string containing a digital signature.
<i>result</i>	A generated signature or a file to store the signature.

SIGNATURE Parameters

The following table describes the status of each return code.

Return Code	Status
0	Success.
1	Unsupported digest algorithm.
2	The data cannot be read.

Return Code Status

Return Code	Status
3	Message digest cannot be obtained.
4	Invalid parameters.
5	Key cannot be read or is in the wrong format / algorithm.
6	Incorrect Password.
7	Signature cannot be generated.
8	Signature cannot be verified.
Return Code Status (Continued)	

Additional Reading

Due to the amount of terminology regarding cryptography in general and SSL in particular, interested readers may refer to the following publications.

“Applied Cryptography”, by Bruce Schneier

“Internet Cryptography”, by Richard E. Smith

“SSL and TLS: Designing and Building Secure Systems”, by Eric Rescorla

Using SSL With UniObjects for Java

Overview of SSL Technology	3-3
Software Requirements	3-4
Setting up Java Secure Socket Extension (JSSE)	3-5
Configuring UOJ to use IBM JSSE	3-6
Configuring the Database Server for SSL	3-7
Creating a Secure Connection	3-9
Direct Connection	3-10
Establishing the Connection	3-12
Proxy Tunneling	3-13
Externally Secure	3-15
Managing Keys and Certificates for a UOJ Client and a Proxy Server	3-20
Importing CA Certificates Into UOJ Client Trustfile	3-20
Generating client certificates.	3-21
Managing Keyfile and Trustfile for the Proxy Server.	3-22

This chapter explains how to use SSL (Secure Socket Layer) with UniObjects for Java (UOJ). The topics covered include:

- “ Overview of SSL Technology”
- “ Software Requirements”
- “ Setting up Java Secure Socket Extension (JSSE)”
- “ Configuring UOJ to use IBM JSSE”
- “ Configuring the Database Server for SSL”
- “ Creating a Secure Connection”

Overview of SSL Technology

Secure Sockets Layer (SSL) is a transport layer protocol that provides a secure channel between two communicating programs over which arbitrary application data can be sent securely. It is by far the most widely deployed security protocol used on the World Wide Web.

Although it is most widely used in applications to secure web traffic, SSL actually is a general protocol suitable for securing a wide variety of other network traffic that is based on TCP, such as FTP and Telnet.

SSL provides server authentication, encryption and message integrity. It optionally also supports client authentication.

This document assumes that users who want to use this facility have some basic knowledge of public key cryptography.

For more information on the implementation of SSL with UniData and UniVerse, refer to *Developing UniBasic Applications* manual for UniData and the *Guide to UniVerse Basic* for UniVerse.

Software Requirements

You must have the following applications installed and configured on the client machine.

- JDK (Java Development Kit) 1.4 or higher
- UniObjects for Java version 2.0.0 or higher

Setting up Java Secure Socket Extension (JSSE)

The java.sun.com web site defines JSSE as a set of Java packages that enable secure Internet communications. JSSE implements a Java version of Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP, and FTP) over TCP/IP.

SSL for UOJ requires an implementation of JSSE to be installed on the client computer as well as the proxy server if one is to be used.

UniObjects for Java ships with the IBM Reference implementation of JSSE, but any implementation from a valid JSSE provider should work. The file that contains the JSSE components is named `ibmjsse.jar` and is located in the archive directory of your UniDK installation, for example: `C:\IBM\UniDK\uoj sdk\lib`.

Configuring UOJ to use IBM JSSE

First, copy `ibmjsse.jar` into the `/lib/ext` directory of your jdk installation or simply edit your `CLASSPATH` environment variable to reference the `ibmjsse.jar` file in the UOJ archive directory specified above.

Second, you will need to add the IBM JSSE provider to the list of security providers in the `java.security` file. This file is located in the jdk installation directory under `/lib/security`. Edit this file with Notepad or another text editor and add the following line:

```
security.provider.N=com.ibm.jsse.JSSEProvider
```

Where N is the number defining the position of the IBM JSSE in the list of security providers. For example, the file would look something like this.

```
security.provider.1=sun.security.provider.Sun
```

```
security.provider.N=com.ibm.jsse.JSSEProvider
```



Note: *If you already have a JSSE security provider installed on the client machine, there is no need to install the IBM JSSE unless you specifically want to use it. If you do decide to use the IBM JSSE, we recommend that you remove any other JSSE security providers to avoid any conflicts or problems.*

Configuring the Database Server for SSL

First, you need to create a Server Security Context Record (SCR).

A SCR contains all SSL related properties necessary for the server to establish a secured connection with an SSL client. The properties include the server's private key, certificate, client authentication flag and strength, and trusted entities. For more information, see *UniBasic Extensions*.

The SCR can be generated by directly calling the UniData or UniVerse Security API from a BASIC program, or alternatively, by invoking UniAdmin.

The SCR is encrypted by a password and saved in a UniData or UniVerse security file with a unique ID. The path, password and ID of the SCR for a UOJ server are important in the following descriptions.

In order to enable SSL support for UOJ on the database server you need to edit two configuration files, *unirpcservices* and *.scrfile*. Both of these files are located in the unishared/unirpc directory. On UNIX systems, you can determine the location of the unishared directory by entering *cat /.unishared*. On Windows platforms, the default location can be found by examining the registry record at HKEY_LOCAL_MACHINE\SOFTWARE\IBM\UniShared.

First, on the database server, edit the unirpcservices file. Open the file with a text editor such as vi on UNIX or Notepad on Windows, and locate the line that corresponds to the UOJ server. The line is similar to the following example:

```
udcs C:\IBM\ud71\bin\udapi_server.exe *TCP/IP 0 3600
```

Append "SCR-ID password" to the end of this line as shown in the following example, where "SCR-ID" is the record ID of your Security Context Record.

```
udcs C:\IBM\ud71\bin\udapi_server.exe *TCP/IP 0 3600 SCR-ID password
```

Now, edit the .scrfile. Refer to the section above to determine its location. This file contains the path to the Security Context Record store, which contains the Security Context Record specified in the "unirpcservices" file. The file format is as follows:

```
service-name    path
```

For example:

```
udcs c:\IBM\ud71\demo
```

Once these files have been edited appropriately, the database server should be properly configured.

Creating a Secure Connection

There are three different modes you can use to establish a secure SSL session with a UniData or UniVerse database server.

- **Direct Connection** - This method is completely secure. In this mode the SSL session is established directly between the UOJ client and the UniData or UniVerse database server.
- **Proxy Tunneling** - This method is completely secure. In this mode, the connection is created through a proxy server. The proxy server provides tunneling for the data exchange between the UOJ client and the UniData or UniVerse database server. Since the proxy server does not decrypt data packets, there is no session multiplexing performed.
- **Externally Secure Proxy** - The security of this method is reliant on the external proxy. In this mode, the externally secure SSL session is established between the UOJ client and an external proxy server. The connection between the proxy server and the UniData or UniVerse database server is not a secure connection. A typical application for this type of connection would be in the case where both the proxy server and UniData or UniVerse database server are behind a firewall. Thus, the unsecured connection between the proxy and database server does not compromise security. In this mode, session multiplexing can be achieved.

The first step is to create a UniSession object by calling the **openSession** method of the **UniJava** object. The signature of the method is shown in the following example.

```
public UniSession openSession(int sslmode) throws  
    UniSessionException
```

The *sslmode* parameter can be one of the following values:

Mode	Option
Direct Connection	UniObjectTokens.SECURE_SESSION
Proxy Tunnel	UniObjectTokens.SECURE_SESSION
Secure External Proxy	UniObjectTokens.EXTERNALLY_SECURE_PROXY_SESSION

Next, determine which of the following connection types you wish to use for the secure connection.

Direct Connection

When creating a secure connection, there are three components that you must consider. They are SSL Socket Factory, Cipher Suites and Keyfile, and Trustfile Parameters. You can define these parameters by creating and setting properties of the **UniSSLDescriptor** object associated with the secure session and setting some system variables.

- **SSL Socket Factory** - Secure Socket Factories encapsulate details for creating and initially configuring secure socket connections. The **SSLSocketFactory** object is a concrete implementation of the abstract **SocketFactory** class provided with JSSE in the `javax.net` package. It acts as a factory for creating secure sockets. You can define your own **SSLSocketFactory** object with the **setSSLSocketFactory** method of the **UniSSLDescriptor** object. If you pass a null parameter to this method, the system defaults will be used. Another way to use the system defaults is to set the **UniSSLDescriptor** object to null by calling the **setSSLDescriptor** method of the **UniSession** object with a null parameter.
- **Cipher Suites** - Define your own available Cipher Suites with the **setEnabledCipherSuites** method of **UniSSLDescriptor**. If you pass a null parameter to this method, the system defaults will be used.
- **Keyfile and Trustfile Parameters** - System Variables must be created to define locations of the keyfile, trustfile and the password to access these files. This step is required for any secure connection.

If **uniojbects.UniSSLDescriptor** is set to null, the system will use the system defaults for **SSLSocketFactory** and default Cipher suites.

Once you have created the session object, to specify your own **SSLSocketFactory** object and/or define available cipher suites, you need to create the **uniojbects.UniSSLDescriptor** using the constructor with the following signature.

```
public UniSSLDescriptor (void)
```

Once created, you need to call the **setSSLSocketFactory** method to set the SSL Socket Factory and **setEnabledCipherSuites** to set the available cipher suites and then pass this object to the session.

Calling the **setSSLSocketFactory** method with the signature shown in the following example will set **SSLSocketFactory**.

```
public void setSSLSocketFactory(SSLSocketFactory sslsf)
```

Calling the **setEnabledCipherSuites** method with the signature shown in the following example sets CipherSuites.

```
public void setEnabledCipherSuites(String [] cs)
```

Whether you specify your own Socket Factory and Cipher Suites or use the system defaults, you still need to specify the system variables for the location and password for the keyfile and the trustfile as shown in the following table:

System Variable	Definition
javax.net.sslTrustStore	Defines the location of the trustfile.
javax.net.sslKeyStore	Defines the location of the keyfile.
javax.net.sslKeyStorePassword	Defines the password for the keyfile.

The trustfile (also called truststore), is a file that holds a set of keys and certificates. In fact, the keyfile (also called keystore) has exactly the same format. The difference between a trustfile and a keyfile is more a matter of function than of a programming construct. The keyfile provides credentials for the secure connection and the trustfile verifies those credentials. The trustfile and keyfile can be, and often are, the same file.

You can use tools such as IBM's ikeyman utility and Sun's keytool to create and maintain the keyfile and trustfile. The Keytool utility is installed with Sun Microsystem's JDK. For more information on keytool, see <http://java.sun.com/products/jdk1.2/docs/tooldocs>. The default location for the trustfile (truststore) is \$JREHOME/lib/security/jssecacerts. If the file does not exist, the system assumes that the trustfile is located under \$JREHOME/lib/security/cacerts. There is no default location for the keyfile (keystore).

Establishing the Connection

Once you have set the secure parameters for the session, you can connect by calling the connect method of the **UniSession** object as you would in any normal, nonsecure session.

The following code example demonstrates how to create a secure Direct Connection with the database server.

Database server Connection Properties:

```
U2 host: localhost
user name:"test"
password:"new.pass"
accountpath: "demo"
```

Security Properties are:

```
keyfile path: "testkeys"
keyfile password: "new.pass"
trustfile path: "testtrust"
trustfile password: "new.pass"
```

```
String U2host = "localhost";
String username = "test";
String password = "new.pass";
String accountpath = "demo";
String keyfilepath = "testkeys";
String keyfilepwd = "new.pass";
String trustfilepath = "testkeys";

// First, let's instantiate our new UOJ application

uvJava = new UniJava();

// Now, let's open up a session

UniSession demoSession =
uvJava.openSession(UniObjectsTokens.SECURE_SESSION);

demoSession.setHostPort(UniRPCTokens.UNIRPC_DEFAULT_PORT );
demoSession.setHostName(U2host );
demoSession.setUserName( username );
demoSession.setPassword( password );
demoSession.setAccountPath( accountpath );

// Now we'll set locations for the keystore and truststore and a password
for the keystore

System.setProperty("javax.net.sslTrustStore", "testtrust");
System.setProperty("javax.net.sslKeyStore", "testkeys");
System.setProperty("javax.net.sslKeyStorePassword.", "new.pass");

demoSession.setSSLDescriptor(null);
demoSession.connect();
```

Proxy Tunneling

The process for using the Proxy Tunneling method is basically the same as the Direct Connection method. The only difference is that the connection is tunnelled through a proxy server which passes messages between the client and database server. There are no additional parameters to configure but the proxy server should be properly configured.

You need to set the `PROXY_SSL_FLAG` parameter in the *uniproxy.config* file to true, so the proxy server will listen for secure connections. See [“Externally Secure”](#) on page 3-14 for more information on editing the *uniproxy.config* file.

The following example demonstrates how to create a secure connection with the database server through a Proxy Tunneling server.

The U2 connection properties are:

```
U2 host: localhost
user name: "test"
password: "new.pass"
accountpath: "demo"
```

Proxy server properties are:

```
Proxy host - localhost
Proxy token - "password1"
```

Security properties are:

```
keyfile path: "testkeys"
keyfile password:
trustfile path: "testkeys"
"new.pass"
```

```

String U2host = "localhost";
String username = "test";

String password = "new.pass";
String accountpath = "demo";

String proxyhost = "localhost";
String proxytoken = "password1";

String keyfilepath = "testkeys";
String keyfilepwd = "new.pass";
String trusfilepath = "testkeys";

int sslmode = UniObjectsTokens.SECURE_SESSION;

// Instantiate our new Uni/Java application
UniJava uvJava = new UniJava();

// First, let's open up a session
UniSession demoSession = uvJava.openSession(sslmode);
demoSession.setHostName( U2Host );
demoSession.setHostPort(UniRPCTokens.UNIRPC_DEFAULT_PORT );
demoSession.setUserName(username );
demoSession.setPassword( password );
demoSession.setAccountPath( accountPath );
demoSession.setProxyHost(proxyhost);
demoSession.setProxyPort(UniRPCTokens.UNIRPC_DEFAULT_PROXY_PORT);
demoSession.setProxyToken(proxytoken);

// Set system variables for locations of the keystore and
truststore and a password for the keystore

System.setProperty("javax.net.sslTrustStore", "testtrust");
System.setProperty("javax.net.sslKeyStore", "testkeys");
System.setProperty("javax.net.sslKeyStorePassword.", "new.pass");

// use default SSLSocketFactory object
demoSession.setSSLDescriptor(null);
demoSession.connect();

```

Externally Secure

This method requires that you define the properties described in the *uniproxy.config* file.

You must set the following parameters for SSL for UOJ configuration.

Parameter	Description
PROXY_SSL_FLAG	This parameter enables or disables externally secure connections. Its value can be <i>true</i> or <i>false</i> . When set to <i>true</i> , the proxy server will start a new thread that listens on PROXY_SSL_PORT for externally secure connections. This parameter must be set to <i>true</i> for both Proxy Tunneling and Externally Secure modes. The default setting is <i>false</i> .
PROXY_SSL_ONLY_FLAG	If this parameter is set to <i>true</i> , the proxy only allows secure connections to pass through to the database server. The default setting is <i>false</i> .
PROXY_SSL_PORT	This parameter defines the port on which the proxy server should listen for externally secure connections.
SSL_KEY_FILE	This parameter specifies the location of the keyfile (keystore).
SSL_TRUST_FILE	This parameter specifies the location of the trustfile (truststore).
SSL_KEY_FILE_TYPE	This parameter specifies the type of the proxy server keyfile type. It can be either <i>JKS</i> or <i>JCEKS</i> . The default value is <i>JKS</i> .
SSL_TRUST_FILE_TYPE	This parameter specifies the type of the proxy server trustfile. It can be either <i>JKS</i> or <i>JCEKS</i> . The default value is <i>JKS</i> .

Parameter	Description
SSL_PWD_METHOD	<p>This parameter defines the method in which password for the keystore is specified. This parameter can take the following values:</p> <p><i>DIRECT</i> - When this value is selected, the password is stored directly in the SSL_KEY_FILE_PWD.</p> <p><i>USER_DEFINED</i> - When you select this value, the parameter, SSL_KEY_FILE_PWD contains a description of how to call a user defined java method that will generate the password. In this case, the value for these properties consists of three fields separated by the underscore character, “_”. The first field is a parameter for the method and should be of type String. The second field is a method name and a third field defines a class name. This mode provides better security for protecting the passwords. However, keep in mind that it may be possible that the password algorithm can be reverse engineered.</p> <p><i>INTERACTIVE</i> - When you select this value, the proxy server prompts the user to enter a password for the keyfile and trustfile interactively during the startup. This mode provides the most password security but cannot support proxy auto-restart.</p>
SSL_KEY_FILE_PWD	This parameter contains information depending on settings defined in the SSL_PWD_METHOD.
SSL_CLIENT_AUTHENTICATION	This parameter specifies whether or not the proxy will ask for a client certificate during the SSL handshake.

The following example demonstrates how to create an Externally Secure connection with the database server.

- The keyfile (keystore) that contains credentials (keys and certificate) for the proxy server is called "testkeys" and is located in the current proxy directory.
- The keyfile type is JKS.
- The proxy server should authenticate all UOJ clients.
- The trustfile (truststore) that contains trusted certificates is called "testtrust" and is located in the current proxy directory.

- The trustfile type is JKS.
- The passwords for the keystore and truststore should be entered interactively.
- The proxy port for listening for externally secure connections is 31452.

The proxy configuration for this example is as follows:

```
PROXY_SSL_FLAG=true  
PROXY_SSL_PORT=31452  
SSL_KEY_FILE=testkeys  
SSL_TRUST_FILE=testtrust  
SSL_KEY_FILE_TYPE=JKS  
SSL_TRUST_FILE_TYPE=JKS  
SSL_PWD_METHOD=INTERACTIVE  
SSL_CLIENT_AUTHENTICATION=true
```

database server: localhost

```
user name:newuser  
password:new.pass  
accountpath: demo
```

Proxy server properties are:

```
Proxy host: localhost  
Proxy token: password1
```

Security properties are:

```
keyfile path: testkeys
keyfile password: new.pass
trustfile path: testtrust

String U2host = localhost;
String username = newuser;

String password = new.pass;
String accountpath = demo;

String proxyhost = localhost;
String proxytoken = password1;

String keyfilepath = testkeys;
String keyfilepwd = new.pass;
String trusfilepath = testkeys;;

int sslmode = UniObjectsTokens.EXTERNALLY_SECURE_PROXY_SESSION;

// Instantiate our new Uni/Java application
UniJava uvJava = new UniJava();

// First, let's open up a sessions
UniSession demoSession = uvJava.openSession(sslmode);

    demoSession.setHostName( U2Host );

demoSession.setHostPort(UniRPCTokens.UNIRPC_DEFAULT_PORT );

demoSession.setUserName(username );
demoSession.setPassword( password );
demoSession.setAccountPath( accountPath );

    demoSession.setProxyHost(proxyhost);

demoSession.setProxyPort(UniRPCTokens.UNIRPC_DEFAULT_SSL_PROXY_PORT);

    demoSession.setProxyToken(proxytoken);

// Set locations for the keystore and truststore and a password for the
keystore
System.setProperty(javax.net.sslTrustStore, testtrust);
System.setProperty(javax.net.sslKeyStore, testkeys);
System.setProperty(javax.net.sslKeyStorePassword, new.pass);

// use default SSLSocketFactory object
    demoSession.setSSLDescriptor(null);

demoSession.connect();
```

Managing Keys and Certificates for a UOJ Client and a Proxy Server

When a server establishes a secure session with a client, it passes its certificate down for authentication. The client usually has a list of trusted certificates that it uses to verify server credentials. If the client cannot verify the server certificate through its trusted certificates, it rejects the connection. Optionally, a server may also require a client to authenticate itself by providing the server with a valid trusted certificate. In the case where the server cannot verify the client certificate, the secure connection is not established. A list of trusted certificates that is used to verify credentials usually resides in a trustfile, and private keys and certificates providing credentials are kept in the keyfile.

A UOJ client should provide the system with a location of trustfile and keyfile and also the keyfile password by setting system properties.

The JDK usually contains a program that works with keyfiles and trustfiles. In Sun Microsystems's implementation of the JDK, this utility is called **keytool**. In IBM's JDK implementation it is called the **ikeman** utility. All examples from this chapter use the **keytool** utility. For a complete description of **keytool** utility, see ["http://java.sun.com/products/jdk/1.4/docs/toddocs/win32/keytool.html"](http://java.sun.com/products/jdk/1.4/docs/toddocs/win32/keytool.html).

Importing CA Certificates Into UOJ Client Trustfile

In general, a server's certificate is issued by a trusted third party called a Certificate Authority (CA), whose certificate (CA certificate) is used to sign the server certificate. In order for a client to verify a server's certificate, the UOJ client should import the trusted server's CA certificate into its trustfile.

Suppose we have a trusted server CA certificate in the file *cacert.pem*, the client's trustfile is called *testtrust*, and the access password for the trustfile is *passphrase*. By executing the following command, you can import the certificate into the trustfile.

```
keytool -import file cacert.pem -keystore testtrust -storepass  
passphrase
```


Generating client certificates

In the case where the database server or the proxy server requires client authentication, the client certificate should be generated and installed into the client's keyfile. Complete the following steps below to generate and install the certificate for the client.

1. Generate a key pair consisting of a public key and a private key. The following command in the **keytool** utility generates an RSA type key pair, as well as a self-signed certificate in the keyfile.

```
keytool -genkey -keystore testkeys -storepass passphrase -keyalg  
RSA
```

2. Create a certificate request. The following command in the **keytool** utility creates a certificate request in the file javacert.req.

```
keytool -certreq keystore testkeys -storepass passphrase -file  
javacert.req
```

3. Send a certificate request to a Certificate Authority (CA). The javacert.req file containing the certificate request should be sent to a valid Certificate Authority that will approve it and send back the certificate chain. We assume that the certificate chain is returned in the file javacert.pem file. A file javacert.pem can be exported to the client keyfile.

If you choose to use the UniData BASIC API to generate certificates for requests, or if the CA described in the previous paragraph returns its CA certificate separately, the server CA certificate should be separately installed into the client's keystore before generated certificates are installed there. The CA Certificate must be imported into the keyfile using an alias, as described in the following example.

```
keytool -import -file cacert.pem -keystore testkeys -storepass  
passphrase -alias ca
```

Where cacert.pem contains the CA certificate and ca is the name of the alias.

4. Replace your own certificate with the newly created CA-signed certificate in the keyfile. The following command in the **keytool** utility will replace the self-signed certificate with the newly generated one.

```
keytool -import -file javacert.pem -keystore testkeys -storepass  
passphrase
```

Managing Keyfile and Trustfile for the Proxy Server.

The keyfile and trustfile for the proxy server should be managed by a standard key and certificate utility, such as Sun Microsystem's **keytool** or IBM's **ikeyman** utility.

Automatic Data Encryption

Encrypted File Types	4-3
Encryption With UniVerse Replication	4-3
Key Store	4-4
How Encryption Works.	4-5
Defining a Master Key	4-7
Changing a Master Key After Data is Encrypted	4-7
UniVerse Encryption Algorithms.	4-8
Encryption Commands	4-9
CREATE.ENCRIPTION.KEY	4-9
DELETE.ENCRIPTION.KEY	4-9
LIST.ENCRIPTION.KEY	4-10
GRANT.ENCRIPTION.KEY	4-10
REVOKE.ENCRIPTION.KEY	4-11
ENCRYPT.FILE	4-12
DECRYPT.FILE	4-16
LIST.ENCRIPTION.FILE	4-21
ACTIVATE.ENCRIPTION.KEY	4-21
DEACTIVATE.ENCRIPTION.KEY	4-22
DISABLE.DECRYPTION	4-22
ENABLE.ENCRIPTION	4-23
UniVerse BASIC Encryption Commands	4-24
ACTIVATEKEY	4-24
DEACTIVATEKEY	4-24
DISABLEDEC	4-25
ENABLEDEC	4-26
STATUS Function Changes	4-26
The encman Utility	4-28

Viewing Audit Trail Information 4-28

Generating a Key Store 4-29

Deleting the Key Store. 4-30

Beta Beta

At this release, automatic data encryption is introduced. With this feature, you can encrypt specified fields or entire records, and UniVerse automatically decrypts the data when accessed by UniVerse or UniVerse BASIC commands. This enhancement includes the following features:

- Defining which fields in the UniVerse file to encrypt
- Automatically encrypt the data you specify when writing the record to the UniVerse file
- Automatically decrypt the data you specify when reading the record from the file
- Key management support
- Audit trail for operations on keys and encrypted files
- Support of Federal Information Processing Standards (FIPS) encryption algorithms, which include popular encryption algorithms DES and AES.

Note: *When using automatic data encryption, performance may degrade due to encryption operations, and more disk space may be required.*



Encrypted File Types

At this release, UniVerse only encrypts hashed files. UniVerse does not encrypt directory files, system log files, dictionary files, or system temporary files. However, UniVerse does encrypt the transaction log file, which contains encrypted data for files that are encrypted.

Encryption With UniVerse Replication

If you are using UniVerse Replication, care must be taken when adding automatic data encryption. If a file that is encrypted is also being replicated, UniVerse transfers encrypted data to the subscribing system. Encryption does not occur on the subscribing system. IBM highly recommends that the encryption configuration be the same on both the publishing and subscribing systems, including the master key, encryption key, encryption file definitions, and the algorithms you specify for encryption. If the configurations are not identical, the replicated data may not be synchronized with the source data, and will not be usable when failover is required.

Key Store

The most important part of an encrypted system is key management. To ensure a fully secure system, UniVerse maintains a key store, with an interface to create keys and reference keys. Keys can be protected through a user-name based access control, and also protected by a password.

The UniVerse key store is protected by a master key. This master key is known only to UniVerse, and is also used in deriving all other keys. After you install UniVerse, you should define a master key, either providing one of your own, or using the UniVerse default.

UniVerse stores the master key and loads it into memory each time UniVerse starts. UniVerse uses the master key to open the key store, and loads keys in the UniVerse work space. UniVerse can also use this master key to recover a key password if it is lost.

How Encryption Works

This section gives an overview of how encryption works on a UniVerse database:

After installing UniVerse, you define a master key. You can define your own master key, or use a UniVerse default. IBM recommends that you define your own master key. UniVerse uses the master key in all operations related to encryption.

When you create a new encryption key, you can choose to protect the key with a password, or rely on the operating system-level user name to control access to the key. You can grant access to the encryption key to other users or groups based on the OS-level account name.

When you create an encrypted file, you must associate a key and an encryption algorithm for each object to encrypt. You can encrypt an entire record or a just a field or fields in the record. UniVerse checks if the user has access permission to the key based on the OS-level user or group ID, then asks for the password if the key is password protected.

During the UniVerse read or write operation, either from UniVerse BASIC, Retrieve, or UniVerse SQL, UniVerse locates the key ID associated with an encrypted field and checks if the key is active. The key is considered active if the user has permission to the key, the key is not password protected, or the key is password protected and the correct password has been provided through the `ACTIVATE.ENCRYPTION.KEY` command or the UniVerse BASIC `ACTIVATEKEY` statement.

If the operation you specify is a read operation and the key is not active, UniVerse returns an error in the UniVerse BASIC `STATUS` command, then presents encrypted data. However, if you disable encryption through the `DISABLE.DECRYPTION` command, UniVerse does not attempt to decrypt the data.

If the operation you specify is a write operation and the key is not active, the encrypted field keeps the original cipher text value, and no new encryption occurs. If the data in the encrypted field is in clear text, the write operation fails.

If you provide your own master key, the encrypted data can only be decrypted on the installed system. If you moved the encrypted data to another system, you must set up the same master key, and the same encryption key(s) with the same password, before you can read the encrypted data.

If you choose to use the UniVerse default master key, if you move the encrypted data and the key store to another UniVerse system, you must set up the same encryption keys with the same passwords before you can decrypt the data.

The following table shows the combination of the master key and the key password and their impact on security level and file portability.

System Master Key / File Encryption Key	No Password	With Password
Default	Minimum Protection. Data can be accessed on another UniVerse system with default master key and encryption key.	Strong Protection. Data can be accessed on another UniVerse system with the default master key and the same encryption key with the same password.
System-Specific (user-defined)	Strong Protection. Data can be accessed on another UniVerse system with the same user-defined master key and encryption key.	Maximum Protection. Data can be accessed on another UniVerse system with the same user-defined master key and the same encryption key and password.

Master Key and Key Password Impact

Defining a Master Key

When you initially install UniVerse, each installation has the same default master key. For a new UniVerse installation, UniVerse displays a message at the end of the installation process to remind you to establish a site-specific master key. For an upgrade installation, UniVerse does not change your master key.

Use the `uvregen` command to define a new master key, as shown in the following example:

```
C:\IBM\UV>uvregen -m new_master_key  
Changing UV master key is DANGEROUS!!!  
Do you really want to change it [No]?Yes
```

If you specify `SYSTEM` for the master key, UniVerse changes the master key to the system default. In order to revert to the system default, you must provide the current master key.

Use `@/full_path` to indicate that the master key is stored in a file, as shown in the following example:

```
@/mysecure/mymaster
```

We recommend that the key file is strongly protected, or removed from the system after the installation is complete and stored in a safe place.

The maximum length of a master key is 64 characters. The master key should be long and difficult to guess.

Changing a Master Key After Data is Encrypted

Once a master key has been used in file encryption, we recommend that you do not change it. All aspects of UniVerse data encryption involves the master key, and changing it makes all previously encrypted data, existing keys, and audit records inaccessible.

If you decide to change the master key, you must first decrypt all encrypted data, save a text copy of your existing audit records, and make sure you can re-create existing encryption keys. If you do not follow these steps, your data will not be accessible after you change the master key.

UniVerse Encryption Algorithms

UniVerse supports the following encryption algorithms:

- AES (AES128, AES192, AES256)
- DES (DES, DES3)
- RC2
- RC4

AES and DES are Federal Information Processing Standards (FIPS) compliant encryption algorithms. Within each group, with the exception of RC4, there are multiple chaining modes (CBC, ECB, OFB, and CFB).

When you encrypt a file, you must specify a specific algorithm to use in encryption. The following table describes valid algorithms for UniVerse decryption:

Type of Encryption Desired	Algorithm to Specify
56-bit key DES encryption	des, des-cbc, des-ecb, des-cfb, or des-ofb
112-bit key ede DES encryption	des_ede, des-ede-cbc, des-ede, des-ede-cfb, or des-ede-ofb
168-bit key ede DES encryption	des3, des_ede3, des_ede3-cbc, des_ede3-cfb, or des_ede3-ofb
128-bit key R2 encryption	rc2, rc2-cbc, rc2-ecb, rc2-cfb, or rc2-ofb
128-bit key RC4 encryption	rc4
128-bit key AES encryption	aes128, aes-128-cbc, aes-128-cfb, or aes-128-ofb
192-bit key AES encryption	aes192, aes-192-cbc, aes-192-cfb, aes-192-ofb
256-bit key AES encryption	aes256, aes-256-cbs, aes-256-ecb, aes-256-cfb, or aes-256-ofb

UniVerse Encryption Algorithms

***Note:** The algorithm specification is case-insensitive.*



Encryption Commands

This section lists commands you can use for encrypting and decrypting your data.

CREATE.ENCRYPTION.KEY

Use the CREATE.ENCRYPTION.KEY command to create an encryption key in the UniVerse key store. We recommend that you create a password for the key.

Syntax

CREATE.ENCRYPTION.KEY *key.id* [*password*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key.id</i>	The encryption key ID.
<i>password</i>	The password for <i>key.id</i> .

CREATE.ENCRYPTION.KEY Parameters

Note: We suggest that the password you create is a phrase that is hard to guess, but easy to remember, using a combination of ASCII characters and digits. If a password contains a space (" "), you must use quotation marks to enclose the password.



DELETE.ENCRYPTION.KEY

Use the DELETE.ENCRYPTION.KEY command to delete a key from a key store. You must be the owner of the file or logged on as root or a UniVerse Administrator to delete an encryption key, and you must provide the correct password. If the key is referenced by any encrypted field or file, deleting the key will fail, unless you specify FORCE.

Syntax

DELETE.ENCRIPTION.KEY [FORCE] *key.id* [*password*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
FORCE	Forces the encryption key to be deleted, even if it is referenced by an encrypted record or field.
<i>key.id</i>	The encryption key to delete.
<i>password</i>	The password for the encryption key to delete.

DELETE.ENCRIPTION.KEY Parameters

LIST.ENCRIPTION.KEY

Use the LIST.ENCRIPTION.KEY command to list the existing keys in the key store. You can also list records in the key store using UniVerse Retrieve commands, such as LIST, LIST.ITEM, SORT, SORT.ITEM, and so forth.

***Note:** The name of the key store file is &KEystore&. Although you can view records from this file using UniVerse Retrieve commands, other UniVerse commands, such as DELETE.FILE and CLEAR.FILE will fail. The ED command will only display encrypted data.*

GRANT.ENCRIPTION.KEY

Use the GRANT.ENCRIPTION.KEY command to grant other users access to the encryption key. When a key is created, only the owner of the key has access. The owner of the key can grant access to other users.

Syntax

GRANT.ENCRIPTION.KEY {PUBLIC | *grantee* {,*grantee*...}}



Parameters

The following table describes each parameter of the syntax.

Parameter	Description
PUBLIC	Grants access to the encryption key to all users on the system.
<i>grantee</i>	<p>Grants access to the encryption key to the <i>grantee</i> you specify. <i>grantee</i> can be a user name, or a group name. If you specify a group name, prefix the name with an asterisk (“*”). When you specify a group name, UniVerse grants access to all users belonging to the group.</p> <p>On Windows platforms, a group name can be a local group or a global group (specified in the form of *Domain\global-group). A user can also be a domain user, specified in the form of Domain\user. In the case of “\” appearing in a group or user name, you should use quotation marks to enclose the name.</p> <p>Grantees cannot grant access to the encryption key to other users.</p> <p><i>Note: To grant access to global users or groups, you must log on as a domain user to creat keys and perform the GRANT operation.</i></p>

GRANT.ENCRIPTION.KEY Parameters

You must grant access to an encryption key even if it does not have password protection if you want other users to use the key. On the other hand, even if you have the correct password for the key, you cannot access it without being granted access.

REVOKE.ENCRIPTION.KEY

Use the REVOKE.ENCRIPTION.KEY command to revoke access to the encryption key from other users. When a key is created, only the owner of the key has access. The owner of the key can revoke access from other users.

Syntax

```
REVOKE.ENCRIPTION.KEY {PUBLIC | grantee {,grantee...}}
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
PUBLIC	Revokes PUBLIC access to the encryption key from all users on the system. For example, if “PUBLIC” access is granted, itis removed. However, this does not revoke individual user or group access that had been granted.
grantee	Revokes access to the encryption key from the <i>grantee</i> you specify. <i>grantee</i> can be a user name, or, on UNIX platforms, a group name. If you specify a group name, prefix the name with an asterisk (“*”). When you specify a group name, UniVerse revokes access from all users belonging to the group. On Windows platforms, a group name can be a local group or a global group (specified in the form of *Domain\global-group). A user can also be a domain user, specified in the form of Domain\user. In the case of “\” appearing in a group or user name, you should use quotation marks to enclose the name. Grantees cannot revoke access to the encryption key from other users.

REVOKE.ENCRYPTION.KEY Parameters

ENCRYPT.FILE

Use the ENCRYPT.FILE command to create a file in which each record is encrypted.

Note: You cannot encrypt an index file.

Syntax

```
ENCRYPT.FILE {<filename> <type> <modulo> <separation> | <30 |
dynamic> parameter [value]...} <USING partition> < {WHOLERECORD
| fieldname },alg,key[,pass] [fieldname,alg,key[,pass]]...>
```



Parameters

Most of the ENCRYPT.FILE parameters are the same as the RESIZE command parameters. If the file you are encrypting is empty, you do not need to specify any of the RESIZE parameters. If the file you are encrypting is not empty, and you know that the file needs resizing because encrypting the file will increase the record size, you should specify the RESIZE parameters.

The following table describes each parameter of the syntax.

Parameter	Description
<i>filename</i>	The UniVerse file name. If you do not specify <i>filename</i> , ENCRYPT.FILE prompts for the name. <i>filename</i> must follow the UniVerse naming conventions. For more information about naming conventions, see “File Naming Conventions” in <i>UniVerse User Reference</i> .
<i>type</i>	The UniVerse file type for the file you are encrypting. Type 1 or type 19 files are not hashed and are usually used to store text files such as BASIC programs. Types 2 through 18 are hashed files. Type 25 is a balanced tree file.
<i>modulo</i>	The modulo for the file you are encrypting. The modulo should be an integer from 1 through 8,388,608 defining the number of groups in the file. UniVerse ignores <i>modulo</i> if you specify a nonhashed or dynamic file type.
<i>separation</i>	The separation for the file you are encrypting. The separation should be an integer from 1 through 8,388,608, specifying the group buffer size is 512-byte blocks. UniVerse ignores <i>separation</i> if you specify a nonhashed or dynamic file type.
30	Encrypts a dynamic file.
dynamic	Encrypts a dynamic file.
USING <i>partition</i>	Specifies the path of the work area that ENCRYPT.FILE will use for creating the necessary temporary files. For example, the following command encrypts SUN.MEMBER as a dynamic file, and creates the temporary files it needs in the partition <i>/u4</i> : >ENCRYPT.FILE SUN.MEMBER DYNAMIC USING /u4 ENCRYPT.FILE moves the files back into the correct directory after encrypting the SUN.MEMBER file.

ENCRYPT.FILE Parameters

Parameter	Description
WHOLERECORD	Specifies to fully encrypt every record in the file.
<i>fieldname,alg,key,pass</i>	<p>Specifies the field name to encrypt, and the algorithm, key, and password to use. You can use a different algorithm and key for each field.</p> <p>If you do not specify a password, but created the key using password protection, UniVerse prompts for the password. If several fields use the same password, you only have to specify it once, at the first field that uses that key.</p>
<i>fieldname</i>	The name of the field to encrypt.
<i>alg</i>	The algorithm to use for encryption. See “ UniVerse Encryption Algorithms ” on page 9 for a list of valid values.
<i>key</i>	The key ID to use for the field encryption.
<i>pass</i>	The password corresponding to the <i>key</i> .

ENCRYPT.FILE Parameters (Continued)

Specify the following parameters only for dynamic files:

Parameter	Description
GENERAL	Specifies the general hashing algorithm for a dynamic file. GENERAL is the default.
SEQ.NUM	Specifies a hashing algorithm suitable for sequential numbers for a dynamic file. Use this hashing algorithm only for records with IDs that are mainly numeric, sequential, and consecutive.
GROUP.SIZE { 1 2 }	Specifies the size of each group in the file, either 1 or 2. 1 specifies a group size of 2048 bytes, which is equivalent to a separation of 4. 2 specifies a group size of 4096 bytes, which is equivalent to a separation of 8. A group size of 2048 (GROUP.SIZE 1) is the default.
MINIMUM.MODULUS <i>n</i>	Specifies the minimum modulo of the file, an integer value greater than 1. This value is also the initial value of the modulo of the dynamic file. A minimum modulo of 1 is the default.

ENCRYPT.FILE Parameters for Dynamic Files

Parameter	Description
SPLIT.LOAD <i>n</i>	Specifies the level at which the file's modulo is increased by 1. SPLIT.LOAD takes a numeric argument indicating the percentage of space allocated for the file. When the data in the file exceeds the specified percentage of the space allocated for the file, the data in one of the groups is divided equally between itself and a new group, to increase the modulo by 1. The default SPLIT.LOAD is 80%.
MERGE.LOAD <i>n</i>	Specifies the level at which the file's modulo is decreased by 1. MERGE.LOAD takes a numeric argument indicating the percentage of space allocated for the file. When the data in the file is less than the specified percentage of the space allocated for the file, the data in the last group of the file is merged with another group, to decrease the modulo by 1. The default MERGE.LOAD is 50%.

ENCRYPT.FILE Parameters for Dynamic Files (Continued)

Parameter	Description
LARGE.RECORD <i>n</i>	Specifies the size of a record considered too large to be included in the primary group buffer, specified as an integer or a percentage. Specified as an integer, the value is the number of bytes a record must contain to be considered a large record. Specified as a percentage, the value is a percentage of the group size. When the size of a record exceeds the specified value, the data for the record is put in an overflow buffer, but the record ID is put in the primary buffer. This method of large record storage increases access speed. The default LARGE.RECORD size is 80%.
RECORD.SIZE <i>n</i>	Calculates the values for group size and large record size based on the value of the estimated average record size specified. The value is your estimate of the average record size for the dynamic file, specified in bytes. RECORD.SIZE does not limit the size of records. If you specify a value for group size (GROUP.SIZE) or for large record size (LARGE.RECORD), those values override the value calculated by RECORD.SIZE.
MINIMIZE.SPACE	Calculates the best amount of space required by the file (at the expense of access time), using the values for the split load, merge load, and large record size. If you specify values for split load, merge load, or large record size, those values override the value calculated by MINIMIZE.SPACE. If you specify MINIMIZE.SPACE and RECORD.SIZE, the value for large record size calculated by MINIMIZE.SPACE is used above the value calculated by RECORD.SIZE.

ENCRYPT.FILE Parameters for Dynamic Files (Continued)

Encrypting a file requires exclusive access to the file, and is very time consuming. During the encryption process, UniVerse creates a temporary file and writes the newly encrypted data to that file. If any errors occur during the encryption process, the command aborts and the original file is left intact.

DECRYPT.FILE

The DECRYPT.FILE command decrypts data in a file or in the fields you specify.

Syntax

```
DECRYPT.FILE {<filename> <type> <modulo> <separation> | <30 |  
dynamic> parameter [value]...} <USING partition> < { WHOLERECORD  
| <fieldname> },key[,pass] [fieldname,key[,pass]]...>
```

Most of the DECRYPT.FILE parameters are the same as the RESIZE command parameters. If the file you are decrypting is empty, you do not need to specify any of the RESIZE parameters. If the file you are decrypting is not empty, and you know that the file needs resizing because decrypting the file will change the record size, you should specify the RESIZE parameters.

The following table describes each parameter of the syntax.

Parameter	Description
<i>filename</i>	The UniVerse file name. If you do not specify <i>filename</i> , DECRYPT.FILE prompts for the name. <i>filename</i> must follow the UniVerse naming conventions. For more information about naming conventions, see “File Naming Conventions” in <i>UniVerse User Reference</i> .
<i>type</i>	The UniVerse file type for the file you are decrypting. Type 1 or type 19 files are not hashed and are usually used to store text files such as BASIC programs. Types 2 through 18 are hashed files. Type 25 is a balanced tree file.
<i>modulo</i>	The modulo for the file you are decrypting. The modulo should be an integer from 1 through 8,388,608 defining the number of groups in the file. UniVerse ignores <i>modulo</i> if you specify a nonhashed or dynamic file type.
<i>separation</i>	The separation for the file you are decrypting. The separation should be an integer from 1 through 8,388,608, specifying the group buffer size is 512-byte blocks. UniVerse ignores <i>separation</i> if you specify a nonhashed or dynamic file type.
30	Decrypts a dynamic file.
dynamic	Decrypts a dynamic file.

DECRYPT.FILE Parameters

Parameter	Description
USING <i>partition</i>	<p>Specifies the path of the work area that DECRYPT.FILE will use for creating the necessary temporary files. For example, the following command decrypts SUN.MEMBER as a dynamic file, and creates the temporary files it needs in the partition /u4:</p> <p>>DECRYPT.FILE SUN.MEMBER DYNAMIC USING /u4</p> <p>DECRYPT.FILE moves the files back into the correct directory after creating the SUN.MEMBER file.</p>
WHOLERECORD	Specifies to fully decrypt every record in the file.
<i>fieldname,key,pass</i>	<p>Specifies the field name to decrypt, and the key, and password to use. You can use a different key for each field.</p> <p>If you do not specify a password, but created the key using password protection, UniVerse prompts for the password. If several fields use the same password, you only have to specify it once, at the first field that uses that key.</p> <p><i>fieldname</i> The name of the field to decrypt.</p> <p><i>key</i> The key ID to use for the field decryption.</p> <p><i>pass</i> The password corresponding to the <i>key</i>.</p>

DECRYPT.FILE Parameters (Continued)

Specify the following parameters only for dynamic files:

Parameter	Description
GENERAL	Specifies the general hashing algorithm for a dynamic file. GENERAL is the default.
SEQ.NUM	Specifies a hashing algorithm suitable for sequential numbers for a dynamic file. Use this hashing algorithm only for records with IDs that are mainly numeric, sequential, and consecutive.
GROUP.SIZE { 1 2 }	<p>Specifies the size of each group in the file, either 1 or 2.</p> <p>1 specifies a group size of 2048 bytes, which is equivalent to a separation of 4. 2 specifies a group size of 4096 bytes, which is equivalent to a separation of 8. A group size of 2048 (GROUP.SIZE 1) is the default.</p>

DECRYPT.FILE Parameters for Dynamic Files

Parameter	Description
MINIMUM.MODULUS <i>n</i>	Specifies the minimum modulo of the file, an integer value greater than 1. This value is also the initial value of the modulo of the dynamic file. A minimum modulo of 1 is the default.
SPLIT.LOAD <i>n</i>	Specifies the level at which the file's modulo is increased by 1. SPLIT.LOAD takes a numeric argument indicating the percentage of space allocated for the file. When the data in the file exceeds the specified percentage of the space allocated for the file, the data in one of the groups is divided equally between itself and a new group, to increase the modulo by 1. The default SPLIT.LOAD is 80%.
MERGE.LOAD <i>n</i>	Specifies the level at which the file's modulo is decreased by 1. MERGE.LOAD takes a numeric argument indicating the percentage of space allocated for the file. When the data in the file is less than the specified percentage of the space allocated for the file, the data in the last group of the file is merged with another group, to decrease the modulo by 1. The default MERGE.LOAD is 50%.

DECRYPT.FILE Parameters for Dynamic Files (Continued)

Parameter	Description
LARGE.RECORD <i>n</i>	Specifies the size of a record considered too large to be included in the primary group buffer, specified as an integer or a percentage. Specified as an integer, the value is the number of bytes a record must contain to be considered a large record. Specified as a percentage, the value is a percentage of the group size. When the size of a record exceeds the specified value, the data for the record is put in an overflow buffer, but the record ID is put in the primary buffer. This method of large record storage increases access speed. The default LARGE.RECORD size is 80%.
RECORD.SIZE <i>n</i>	Calculates the values for group size and large record size based on the value of the estimated average record size specified. The value is your estimate of the average record size for the dynamic file, specified in bytes. RECORD.SIZE does not limit the size of records. If you specify a value for group size (GROUP.SIZE) or for large record size (LARGE.RECORD), those values override the value calculated by RECORD.SIZE.
MINIMIZE.SPACE	Calculates the best amount of space required by the file (at the expense of access time), using the values for the split load, merge load, and large record size. If you specify values for split load, merge load, or large record size, those values override the value calculated by MINIMIZE.SPACE. If you specify MINIMIZE.SPACE and RECORD.SIZE, the value for large record size calculated by MINIMIZE.SPACE is used above the value calculated by RECORD.SIZE.

DECRYPT.FILE Parameters for Dynamic Files (Continued)

If the encrypted file was created using the WHOLERECORD keyword, you should specify WHOLERECORD when decrypting the file. If the file was not encrypted using the WHOLERECORD keyword, do not specify WHOLERECORD when decrypting the file.

LIST.ENCRYPTION.FILE

Use the LIST.ENCRYPTION.FILE command to display encryption configuration data, such as the fields that are encrypted, the algorithms used, and so forth. This command also displays the fields for which decryption is currently disabled.

Syntax

LIST.ENCRYPTION.FILE *filename*

ACTIVATE.ENCRYPTION.KEY

Use the ACTIVATE.ENCRYPTION.KEY command to activate a key. It is necessary to activate a key if you want to supply a password for key protection.

Syntax

ACTIVATE.ENCRYPTION.KEY *key.id password* [ON *<hostname>*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key.id</i>	The key ID to activate.
<i>password</i>	The password corresponding to <i>key.id</i> .
ON <i>hostname</i>	The name of the remote host on which you want to activate the encryption key.

ACTIVATE.ENCRYPTION.KEY Parameters

Note: You can activate only keys with password protection with this command. Keys that do not have password protection are automatically activated. Also, you can activate only keys to which you are granted access.

DEACTIVATE.ENCRYPTION.KEY

Use the DEACTIVATE.ENCRYPTION.KEY command to deactivate one or more encryption keys. This command is useful to deactivate keys to make your system more secure.



Syntax

DEACTIVATE.ENCRIPTION.KEY *key.id password* [ON *<hostname>*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key.id</i>	The key ID to deactivate.
<i>password</i>	The password corresponding to <i>key.id</i> .
ON <i>hostname</i>	The name of the remote host on which you want to deactivate the encryption key.

DEACTIVATE.ENCRIPTION.KEY Parameters

Note: *You can deactivate only keys with password protection with this command. Keys that do not have password protection are automatically activated and cannot be deactivated.*

DISABLE.DECRYPTION

Use the DISABLE.DECRYPTION command to turn off decryption on a field you specify.

Syntax

DISABLE.DECRYPTION *filename <field_list>*



Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>filename</i>	The name of the file on which you want to disable description.
<i>field_list</i>	A comma-separated list of fields for which you want to disable decryption. Do not enter spaces between the field names.

DISABLE.DECRYPTION Parameters

ENABLE.ENCRYPTION

Use the ENABLE.ENCRYPTION command to activate encryption on specific fields in a file.

Syntax

ENABLE.ENCRYPTION *filename* <*field_list*>

Parameters

The following table describes each parameter of the syntax..

Parameter	Description
<i>filename</i>	The name of the file on which you want to enable encryption.
<i>field_list</i>	A comma-separated list of fields for which you want to enable encryption. Do not enter spaces between the field names.

ENABLE.ENCRYPTION Parameters

UniVerse BASIC Encryption Commands

This section describes the UniVerse BASIC commands for use with encryption and decryption.

ACTIVATEKEY

Use the ACTIVATEKEY command to activate a key. It is necessary to activate a key if you want to supply a password for key protection.

Syntax

ACTIVATEKEY <key.id>, <password> [ON <hostname>]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key.id</i>	The key ID to activate.
<i>password</i>	The password corresponding to <i>key.id</i> .
ON <i>hostname</i>	The name of the remote host on which you want to activate the encryption key.

ACTIVATEKEY Parameters

***Note:** You can activate only keys with password protection with this command. Keys that do not have password protection are automatically activated. Also, you can activate only keys to which you are granted access.*

DEACTIVATEKEY

Use the DEACTIVATEKEY command to deactivate one or more encryption keys. This command is useful to deactivate keys to make your system more secure.



Syntax

DEACTIVATEKEY <key.id>, <password> [ON <hostname>]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>key.id</i>	The key ID to deactivate.
<i>password</i>	The password corresponding to <i>key.id</i> .
ON <i>hostname</i>	The name of the remote host on which you want to deactivate the encryption key.

DEACTIVATEKEY Parameters

Note: You can deactivate only keys with password protection with this command. Keys that do not have password protection are automatically activated and cannot be deactivated.

DISABLEDEC

Use the DISABLEDEC command to turn off decryption on a file or fields you specify.

Syntax

DISABLEDEC <filename> [, <multilevel-filename>], <field_list>



Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>filename</i>	The name of the file on which you want to disable decryption.
<i>field_list</i>	A comma-separated list of fields for which you want to disable decryption. Do not enter spaces between the field names.

DISABLEDEC Parameters

ENABLEDEC

Use the ENABLEDEC command to activate decryption on a file or fields you specify.

Syntax

ENABLEDEC <*filename*> [, <*multilevel-filename*>], <*field_list*>

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>filename</i>	The name of the file on which you want to enable decryption.
<i>field_list</i>	A comma-separated list of fields for which you want to enable decryption. Do not enter spaces between the field names.

ENABLEDEC Parameters

STATUS Function Changes

The following changes have been made to the UniVerse BASIC STATUS function:

- For UniVerse BASIC READ statements, STATUS() returns 5 to indicate that an encryption error occurred during the READ operation.

- For UniVerse BASIC WRITE statements, STATUS() returns -9 to indicate that an encryption error occurred during the WRITE operation.
- When an encryption error occurs, a READ/WRITE statement will execute statements following the ELSE clause, if an ELSE clause is specified.

The encman Utility

The encman utility enables you to manage data encryption. You can view audit trail information, create a key store, or delete a key store through this utility.

Viewing Audit Trail Information

Use the encman -audit command to view audit trail information.

Syntax

```
encman [ [-audit] [-b date] [-a date] [-u username] [-o operation] [-f]
[-backup <file>] [-use <file>]]
```

The following table describes each parameter of the syntax.

Parameter	Description
-b <i>date</i>	Displays audit trail data before the date you specify. Enter the date in the mm/dd/yyyy format.
-a <i>date</i>	Displays audit trail data after the date you specify. Enter the date in the mm/dd/yyyy format.
-u <i>username</i>	Displays audit trail data for the user name you specify. You can specify multiple users, for example, -u user1 -u user2.

encman -audit Parameters

Parameter	Description
-o <i>operation</i>	Displays audit trail data for the operation you specify. You can specify multiple operations. Valid operations are: <ul style="list-style-type: none"> ■ CREATE – Creating encryption key ■ DELETE – Deleting encryption key ■ GRANT – Granting key access ■ REVOKE – Revoking key access ■ ACTIVATE – Activating encryption key ■ DEACTIVT – Deactivating encryption key ■ ENABLE – Enabling encryption key ■ DISABLE – Disabling encryption key ■ ENCRYPT – Encrypting a file ■ DECRYPT – Decrypting a file ■ RMKEYSTR – Deleting Key Store
-f	Displays only failed operations.
-backup < <i>file</i> >	Backs up the current audit file to the < <i>file</i> > you specify, then clears the audit file.
-use < <i>file</i> >	Displays data in the < <i>file</i> > you specify, rather than the current audit file.

encman -audit Parameters (Continued)

Generating a Key Store

To generate a key store, use the -genkeystore option.

Syntax

```
encman [ [-genkeystore] [-n] ]
```

The following table describes each parameter of the syntax.

Parameter	Description
-n	Specifies to not create the &ENCINFO& file.

encman -genkeystore Parameters

Deleting the Key Store

To delete the current key store, use the -delkeystore option.

Syntax

encman [[-delkeystore] [-f]]

The following table describes the parameter of the syntax:

Parameter	Description
-f	Deletes the key store without prompting for confirmation. <i>Note: Using this operation is dangerous. If you have encrypted files, data cannot be retrieved unless you recreate the keystore and keys used by these files.</i>
encman -delkeystore Parameter	

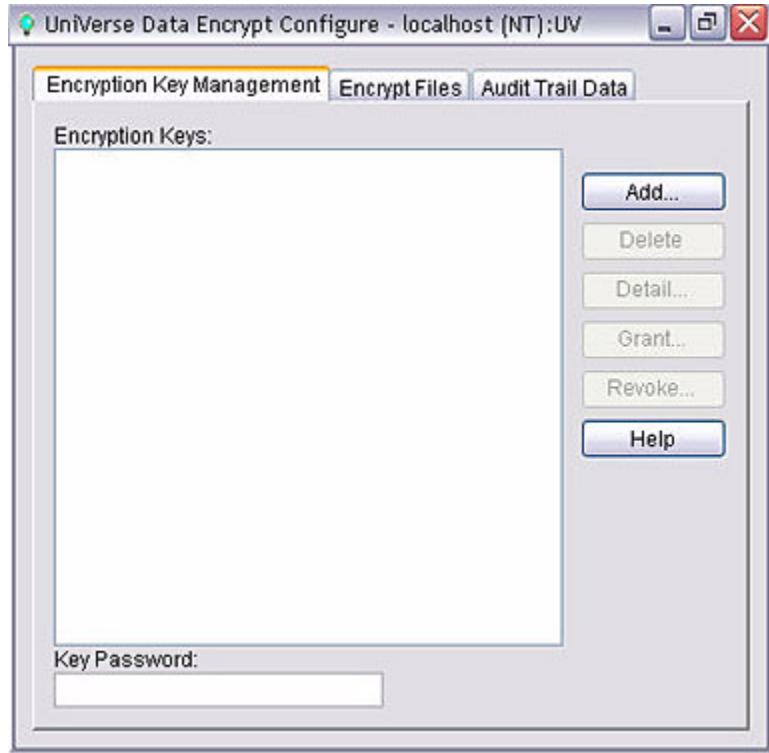
Using UniAdmin for Data Encryption

Using UniAdmin for Encryption	5-2
Adding an Encryption Key	5-3
Deleting an Encryption Key	5-3
Viewing Encryption Key Details	5-4
Granting Privileges.	5-5
Revoking Privileges	5-6
Encrypting a File	5-7
Decrypting a File	5-10
Listing Encryption Information	5-13
Viewing Audit Information	5-15

Using UniAdmin for Encryption

You can use UniAdmin to manage data encryption on your system.

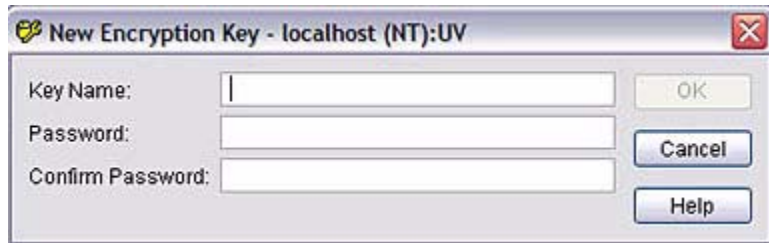
From the UniAdmin main window, select **Data Encrypt Configure**. The **UniVerse Data Encrypt Configure** dialog box appears, as shown in the following example:



If you want to Delete, Grant privileges

Adding an Encryption Key

To create an encryption key, click **Add**. The **New Encryption Key** dialog box appears, as shown in the following example:


A screenshot of the 'New Encryption Key' dialog box. The title bar reads 'New Encryption Key - localhost (NT):UV'. The dialog contains three input fields: 'Key Name:', 'Password:', and 'Confirm Password:'. To the right of these fields are three buttons: 'OK', 'Cancel', and 'Help'.

Enter the name of the encryption key in the **Key Name** box. Although not required, you can enter a password for the new key in the **Password** box. Reenter the password in the **Confirm Password** box.

After you create the encryption key, it appears in the **Encryption Keys** area of the **Data Encrypt Configure** dialog box.

Deleting an Encryption Key

To delete an encryption key, from the **Data Encrypt Configure** dialog box, click the encryption key you want to delete, then click **Delete**. The following dialog box appears:

A screenshot of the 'Key Management' dialog box. The title bar reads 'Key Management - localhost (NT...'. The dialog contains a question mark icon in a speech bubble and the text 'Do you really want to delete key test1'. At the bottom are two buttons: 'Yes' and 'No'.

If you want to delete the encryption key, click **Yes**. If not, click **No**.

Viewing Encryption Key Details

To view details about an encryption key, click the encryption key for which you want to view details from the **Data Encrypt Configure** dialog box, then click **Detail**. The **Encryption Key Details** dialog box appears, as shown in the following example:

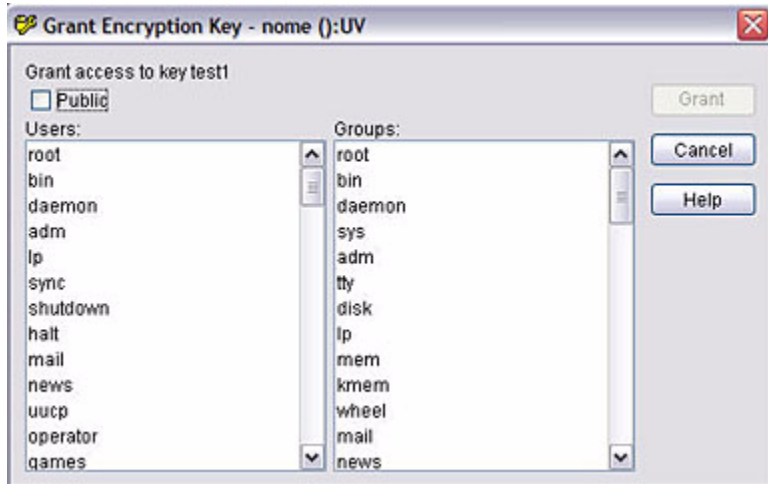


The **Encryption Key Details** dialog box displays the following information about an encryption key:

- **Key Name** – The name of the encryption key.
- **Creator** – The user ID of the user who created the key.
- **Date** – The date the encryption key was created.
- **Time** – The time the encryption key was created.
- **Grantees** – The users or groups who have access to the encryption key.
- **References** – The files and fields for which the encryption key is being used.

Granting Privileges

To grant privileges to the encryption key to a user or group, click **Grant**. The **Grant Encryption Key** dialog box appears, as shown in the following example:



To grant Public privileges, click the **Public** check box.

To grant privileges to individual users, click the user ID of each user for which you want to grant privileges, or click the group ID for which you want to grant privileges, then click **Grant**. To select multiple users or groups, hold the CTRL key down while selecting the users or groups.



Note: You can only grant privileges to Public on Windows platforms.

Revoking Privileges

To revoke privileges from an encryption key from a user or group, click **Revoke**. The **Revoke Encryption Privilege** dialog box appears, as shown in the following example:



To revoke privileges from Public users, click the **Public** check box.

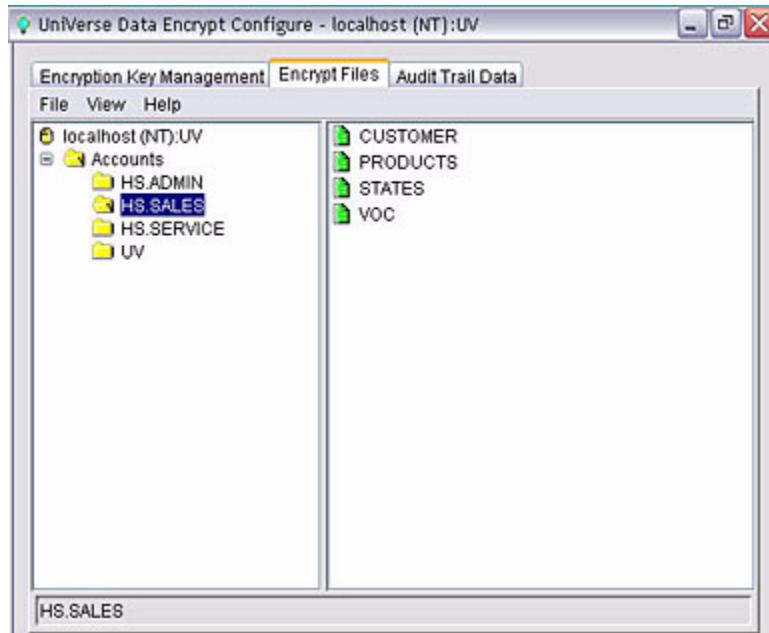
To revoke privileges from individual users, click the user ID of each user for which you want to revoke privileges, or click the group ID for which you want to revoke privileges, then click **Revoke**. To select multiple users or groups, hold the CTRL key down while selecting the users or groups.



Note: You can only revoke privileges from Public on Windows platforms.

Encrypting a File

To encrypt a file or fields in a file, check the **Encrypt File** tab. A window similar to the following example appears:



In the **Accounts** area of the screen, click the account where you want to encrypt files. With the right mouse button, click the file in which you want to encrypt fields.

The **Encrypt File** dialog box appears, as shown in the following example:

Encrypt File - localhost (NT):UV

Data File: C:\IBM\UWHS.SALES\CUSTOMER

Dict File: C:\IBM\UWHS.SALES\ID_CUSTOMER

Parameters:

☐ Whole record

Encrypt Info

Algorithm: AES192

Key: test1

Password:

Apply

Fields Encryption Info:

	@ID	Algorithm	Key	Password
<input type="checkbox"/>	@ID			
<input type="checkbox"/>	ADDR1			
<input type="checkbox"/>	ADDR2			
<input type="checkbox"/>	BUY_DATE			
<input type="checkbox"/>	CITY			
<input type="checkbox"/>	COMPANY			
<input type="checkbox"/>	CUSTID			
<input type="checkbox"/>	FNAME			
<input type="checkbox"/>	LNAME			
<input type="checkbox"/>	PAID_DATE			
<input type="checkbox"/>	PHONE			
<input type="checkbox"/>	PRICE			

Set...
Unset
Unset All

Encrypting an Entire Record

Define the following information to encrypt an entire record:

- **Data File** – The full path to the data file where you want to encrypt data.
- **Dict File** – The full path to the dictionary file where you want to encrypt data.
- **Parameters** – The parameters to use when encrypting the file. For a list of valid parameters, see [ENCRYPT.FILE](#) in Chapter 4, “Automatic Data Encryption.”
- **Whole record** – If you want to encrypt each field in the record, click the **Whole record** check box.

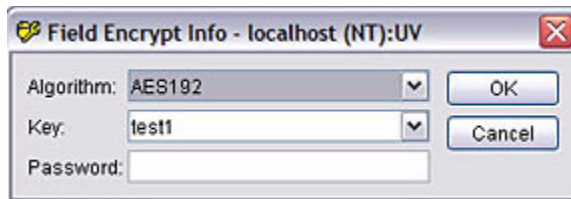
- **Encrypt Info** – Define the following information in the **Encrypt Info** area if you are encrypting an entire record:
 - **Algorithm** – Enter the algorithm to use for encrypting the record. For a list of valid algorithms, see [UniVerse Encryption Algorithms](#) in Chapter 4, “Automatic Data Encryption.”
 - **Key** – Select the encryption key you want to use when encrypting the data from the **Key** list.
 - **Password** – Enter the password corresponding the encryption key, if one exists.

Click **Apply**. UniVerse encrypts every field for every record in the file.

Encrypting Specific Fields In a Record

Define the following information to encrypt specific fields in a record:

- **Data File** – The full path to the data file where you want to encrypt data.
- **Dict File** – The full path to the dictionary file where you want to encrypt data.
- **Parameters** – The parameters to use when encrypting the file. For a list of valid parameters, see [ENCRYPT.FILE](#) in Chapter 4, “Automatic Data Encryption.”
- **Fields Encryption Info** – Click the name of the field you want to encrypt, then click **Set**. The **Field Encrypt Info** dialog box appears, as shown in the following example:

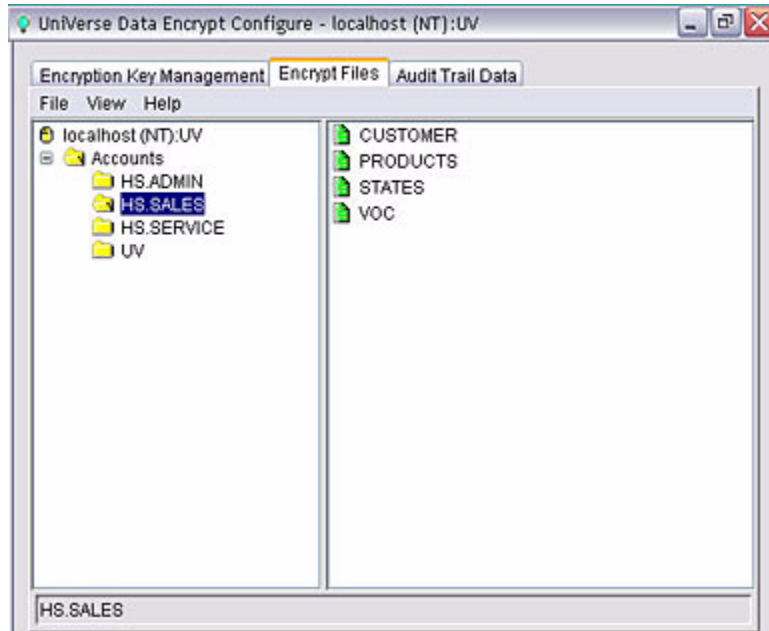


- **Algorithm** – Select the algorithm to use when encrypting the field. For a list of valid algorithms, see [UniVerse Encryption Algorithms](#) in Chapter 4, “Automatic Data Encryption.”
- **Key** – Select the key to use when encrypting the file.
- **Password** – Enter the password corresponding to the key.

When you have defined all the fields you want to encrypt, click **Encrypt**.

Decrypting a File

To decrypt a file or fields in a file, check the **Encrypt Files** tab. A window similar to the following example appears:



In the **Accounts** area of the screen, click the account where you want to decrypt files. With the right mouse button, click the file in which you want to decrypt fields.

The **Decrypt File** dialog box appears, as shown in the following example:

Decrypt File - localhost (NT):UV

Data File: C:\BIM\UVHS.SALES\CUSTOMER

Dict File: C:\BIM\UVHS.SALES\DICTIONARY.DICT

Parameters:

☐ Whole record

Encrypt Info

Algorithm: AES192

Key: test1

Password:

Apply

Fields Encryption Info:

	@ID	Algorithm	Key	Password
<input type="checkbox"/>	@ID			
<input type="checkbox"/>	ADDR1			
<input type="checkbox"/>	ADDR2			
<input type="checkbox"/>	BUY_DATE			
<input type="checkbox"/>	CITY			
<input type="checkbox"/>	COMPANY			
<input type="checkbox"/>	CUSTID			
<input type="checkbox"/>	FNAME			
<input type="checkbox"/>	LNAME			
<input type="checkbox"/>	PAID_DATE			
<input type="checkbox"/>	PHONE			
<input type="checkbox"/>	PRICE			
<input type="checkbox"/>	PRODID			
<input type="checkbox"/>	SAL			

Set... Unset UnsetAll

Decrypting an Entire Record

Define the following information to decrypt an entire record:

- **Data File** – The full path to the data file where you want to decrypt data.
- **Dict File** – The full path to the dictionary file where you want to decrypt data.
- **Parameters** – The parameters to use when decrypting the file. For a list of valid parameters, see [DECRYPT.FILE](#) in Chapter 4, “Automatic Data Encryption.”
- **Whole record** – If you want to decrypt each field in the record, click the **Whole record** check box.

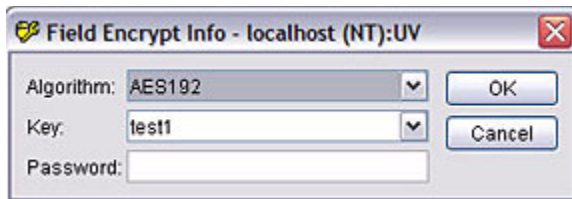
- **Encrypt Info** – Define the following information in the **Encrypt Info** area if you are decrypting an entire record:
 - **Algorithm** – Enter the algorithm to use for encrypting the record. For a list of valid algorithms, see [UniVerse Encryption Algorithms](#) in Chapter 4, “Automatic Data Encryption.”
 - **Key** – Select the encryption key you want to use when decrypting the data from the **Key** list.
 - **Password** – Enter the password corresponding the encryption key, if one exists.

Click **Apply**. UniVerse decrypts every field for every record in the file.

Decrypting Specific Fields In a Record

Define the following information to decrypt specific fields in a record:

- **Data File** – The full path to the data file where you want to decrypt data.
- **Dict File** – The full path to the dictionary file where you want to decrypt data.
- **Parameters** – The parameters to use when decrypting the file. For a list of valid parameters, see [DECRYPT.FILE](#) in Chapter 4, “Automatic Data Encryption.”
- **Fields Encryption Info** – Click the name of the field you want to decrypt, then click **Set**. The **Field Encrypt Info** dialog box appears, as shown in the following example:

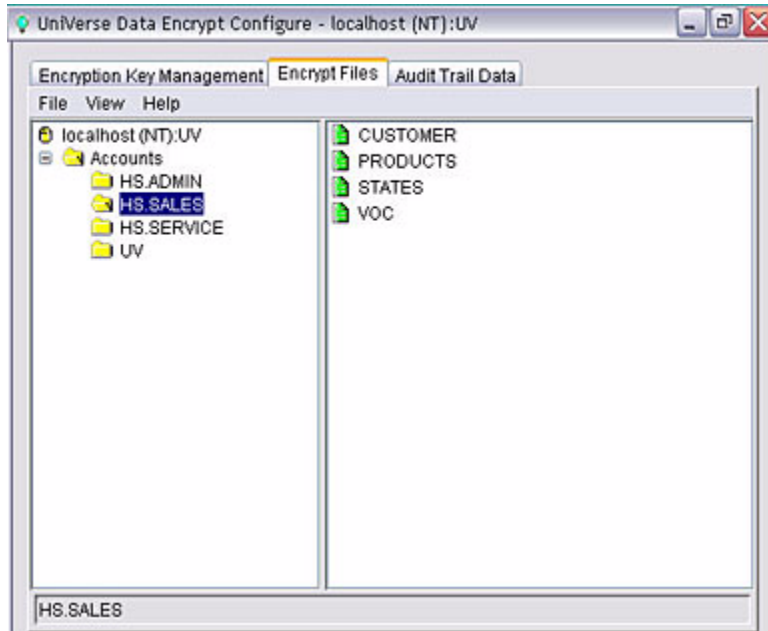


- **Algorithm** – Select the algorithm to use when decrypting the field. For a list of valid algorithms, see [UniVerse Encryption Algorithms](#) in Chapter 4, “Automatic Data Encryption.”
- **Key** – Select the key to use when decrypting the file.
- **Password** – Enter the password corresponding to the key.

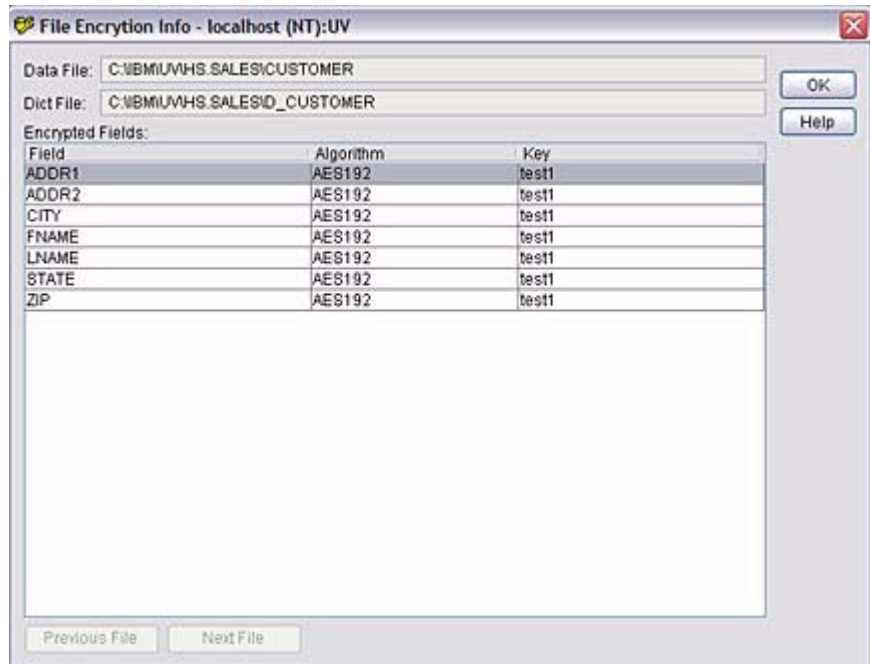
When you have defined all the fields you want to decrypt, click **Decrypt**.

Listing Encryption Information

To list encryption information for a file, click the **Encrypt Files** tab. A window similar to the following example appears:



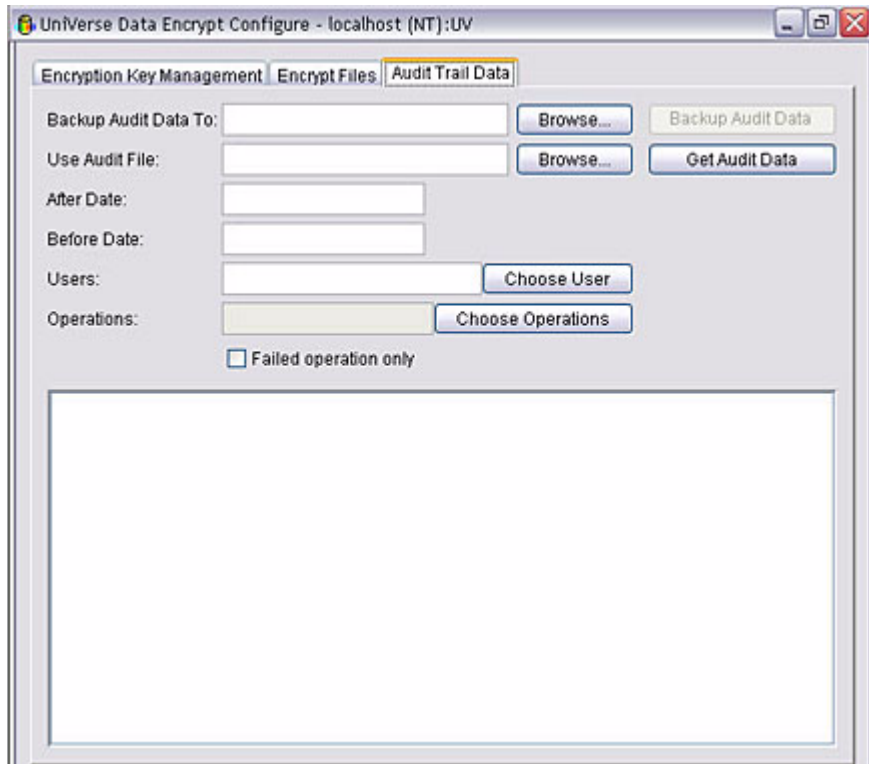
In the **Accounts** area of the screen, click the account where you want to view encryption information. With the right mouse button, click the file for which you want to view encryption information, then click **List Encrypt Info**. A dialog box similar to the following example appears:



Each field that has been encrypted is listed in the **Encrypted Fields** area of the dialog box. If the entire record is encrypted, Whole record appears under the **Field** column.

Viewing Audit Information

To view audit information, from the **UniVerse Data Encrypt Configure** dialog box, click **Audit Trail Data**. A dialog box similar to the following example appears:



The **Audit Trail Data** dialog box offers the following options:

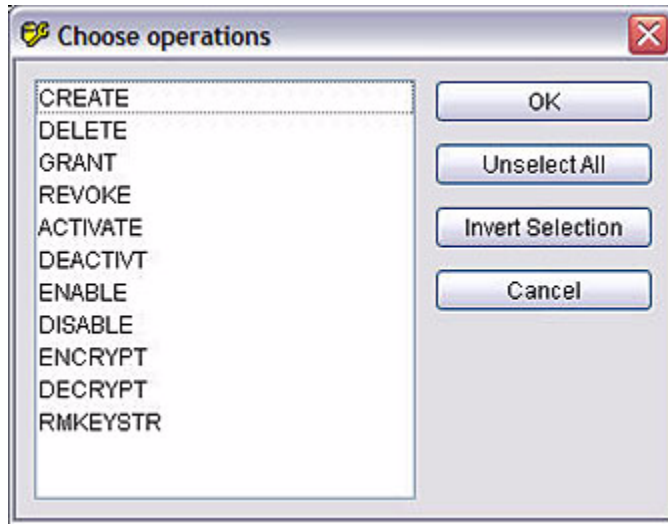
- **Backup Audit Data To** – If you want to backup the current audit file, enter the path to the file where you want to back up the file, or click **Browse** to select the file. After backing up the file, the current audit file is cleared.
- **Use Audit File** – If you want to display audit data located in a file different from the current audit file, enter the full path to the file you want to display, or click **Browse** to select the file.
- **After Date** – UniAdmin will display the audit information after the date you specify. Enter the date in the mm/dd/yyyy format.

- **Before Date** – UniAdmin will display the audit information before the date you specify. Enter the date in the mm/dd/yyyy format.
- **Users** – UniAdmin will display audit trail data for the users you specify. Click **Choose User**. A dialog box similar to the following example appears:



To select multiple users, hold the CTRL key down while selecting the desired users.

- **Operations** – UniAdmin will display audit trail information on the operation you specify. Click **Choose Operations**. The following dialog box appears:



To select multiple operations, hold the CTRL key down while selecting the desired operation. Valid operations are:

- **CREATE** – Creating encryption key
- **DELETE** – Deleting encryption key
- **GRANT** – Granting key access
- **REVOKE** – Revoking key access
- **ACTIVATE** – Activating encryption key
- **DEACTIVT** – Deactivating encryption key
- **ENABLE** – Enabling encryption key
- **DISABLE** – Disabling encryption key
- **ENCRYPT** – Encrypting a file
- **DECRYPT** – Decrypting a file
- **RMKEYSTR** – Deleting Key Store

If you only want to display audit trail data for failed operations, select the **Failed operations only** check box.

Click **Get Audit Data**. UniAdmin displays the audit trail data for the criteria you specified, as shown in the following example:

The screenshot shows the 'Universe Data Encrypt Configure' window with the 'Audit Trail Data' tab selected. The window contains several input fields and buttons for configuring audit data retrieval.

Fields and Buttons:

- Backup Audit Data To:** Text input field with a **Browse...** button.
- Use Audit File:** Text input field with a **Browse...** button.
- After Date:** Text input field.
- Before Date:** Text input field.
- Users:** Text input field containing 'cgustafs' with a **Choose User** button.
- Operations:** Text input field with a **Choose Operations** button.
- ☐ **Failed operation only**
- Backup Audit Data** (disabled button)
- Get Audit Data** (active button)

Audit Trail Data Log:

```
07/07/2006 15:27:09 ENCRYPT cgustafs 0 file <KEYSTORE>, fields WHOLERECORD  
1 audit record displayed
```