Ce logiciel et la documentation qui l’accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d’utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le dé assembler ou de le décompiler, excepté à des fins d’interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu’elles soient exemptes d’erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l’accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l’utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s’applique:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are “commercial computer software” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n’est pas conçu ni n’est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d’Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d’Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée d'The Open Group.

Ce logiciel ou matériel et la documentation qui l’accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l’accès à des contenus, produits ou services tiers, ou à leur utilisation.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>15</td>
</tr>
<tr>
<td><strong>1 Oracle Solaris Security for Developers (Overview)</strong></td>
<td>19</td>
</tr>
<tr>
<td>Overview of Oracle Solaris Security Features for Developers</td>
<td>19</td>
</tr>
<tr>
<td>System Security</td>
<td>19</td>
</tr>
<tr>
<td>Network Security Architecture</td>
<td>21</td>
</tr>
<tr>
<td><strong>2 Developing Privileged Applications</strong></td>
<td>23</td>
</tr>
<tr>
<td>Privileged Applications</td>
<td>23</td>
</tr>
<tr>
<td>About Privileges</td>
<td>24</td>
</tr>
<tr>
<td>How Administrators Assign Privileges</td>
<td>24</td>
</tr>
<tr>
<td>How Privileges Are Implemented</td>
<td>24</td>
</tr>
<tr>
<td>Compatibility Between the Superuser and Privilege Models</td>
<td>26</td>
</tr>
<tr>
<td>Privilege Categories</td>
<td>26</td>
</tr>
<tr>
<td>Programming with Privileges</td>
<td>27</td>
</tr>
<tr>
<td>Privilege Data Types</td>
<td>27</td>
</tr>
<tr>
<td>Privilege Interfaces</td>
<td>28</td>
</tr>
<tr>
<td>Privilege Coding Example</td>
<td>30</td>
</tr>
<tr>
<td>Guidelines for Developing Privileged Applications</td>
<td>33</td>
</tr>
<tr>
<td>About Authorizations</td>
<td>34</td>
</tr>
<tr>
<td><strong>3 Writing PAM Applications and Services</strong></td>
<td>37</td>
</tr>
<tr>
<td>Introduction to the PAM Framework</td>
<td>37</td>
</tr>
<tr>
<td>PAM Service Modules</td>
<td>38</td>
</tr>
<tr>
<td>PAM Library</td>
<td>39</td>
</tr>
<tr>
<td>PAM Authentication Process</td>
<td>39</td>
</tr>
<tr>
<td>Requirements for PAM Consumers</td>
<td>40</td>
</tr>
</tbody>
</table>
# Contents

PAM Configuration ................................................................................................................ ............. 40
Writing Applications That Use PAM Services ................................................................................. 41
  A Simple PAM Consumer Example ......................................................................................... 41
  Other Useful PAM Functions .................................................................................................. 44
Writing Conversation Functions ................................................................................................... 45
Writing Modules That Provide PAM Services ................................................................................. 49
  Requirements for PAM Service Providers ............................................................................. 49
  Sample PAM Provider Service Module .................................................................................. 50

## 4 Writing Applications That Use GSS-API .................................................................................. 55
  Introduction to GSS-API ............................................................................................................ 55
    Application Portability With GSS-API .................................................................................. 56
    Security Services in GSS-API .............................................................................................. 57
    Available Mechanisms in GSS-API ..................................................................................... 57
    Remote Procedure Calls With GSS-API ............................................................................... 57
    Limitations of GSS-API ....................................................................................................... 58
    Language Bindings for GSS-API ......................................................................................... 59
    Where to Get More Information on GSS-API ...................................................................... 59
  Important Elements of GSS-API .............................................................................................. 59
    GSS-API Data Types ............................................................................................................ 59
    GSS-API Status Codes ........................................................................................................ 68
    GSS-API Tokens .................................................................................................................. 69
  Developing Applications That Use GSS-API ............................................................................. 71
    Generalized GSS-API Usage ............................................................................................... 71
    Working With Credentials in GSS-API ............................................................................... 72
    Working With Contexts in GSS-API .................................................................................... 73
    Sending Protected Data in GSS-API .................................................................................... 83
    Cleaning Up a GSS-API Session .......................................................................................... 91

## 5 GSS-API Client Example ......................................................................................................... 93
  GSSAPI Client Example Overview .......................................................................................... 93
  GSSAPI Client Example Structure .......................................................................................... 93
  Running the GSSAPI Client Example ...................................................................................... 94
  GSSAPI Client Example: main() Function ........................................................................... 95
  Opening a Connection With the Server .................................................................................. 96
Figures

FIGURE 3–1  PAM Architecture ................................................................. 38
FIGURE 4–1  GSS-API Layer ................................................................. 56
FIGURE 4–2  RPCSEC_GSS and GSS-API ........................................ 58
FIGURE 4–3  Internal Names and Mechanism Names .......................... 62
FIGURE 4–4  Comparing Names (Slow) ............................................. 64
FIGURE 4–5  Comparing Names (Fast) ............................................... 66
FIGURE 4–6  Exporting Contexts: Multithreaded Acceptor Example .... 82
FIGURE 4–7  gss_get_mic() versus gss_wrap() ................................. 84
FIGURE 4–8  Message Replay and Message Out-of-Sequence .......... 88
FIGURE 4–9  Confirming MIC Data ....................................................... 89
FIGURE 4–10 Confirming Wrapped Data ........................................... 91
FIGURE 7–1  SASL Architecture .......................................................... 124
FIGURE 7–2  SASL Life Cycle .............................................................. 128
FIGURE 7–3  SASL Session Initialization ........................................... 131
FIGURE 7–4  SASL Authentication: Sending Client Data .................. 132
FIGURE 7–5  SASL Authentication: Processing Server Data ............ 134
FIGURE 8–1  Overview of the Oracle Solaris Cryptographic Framework .. 149
FIGURE 10–1  Smart Card Framework .............................................. 182
FIGURE B–1  Major-Status Encoding .............................................. 218
Tables

TABLE 2–1 Interfaces for Using Privileges ................................................................. 28
TABLE 2–2 Privilege Set Transition ........................................................................... 32
TABLE B–1 GSS-APICallingErrors ........................................................................ 218
TABLE B–2 GSS-API Routine Errors ....................................................................... 219
TABLE B–3 GSS-API Supplementary Information Codes ....................................... 220
TABLE B–4 Channel Binding Address Types ............................................................ 223
TABLE B–5 Kerberos v5 Status Codes 1 ................................................................. 227
TABLE B–6 Kerberos v5 Status Codes 2 .................................................................. 228
TABLE B–7 Kerberos v5 Status Codes 3 .................................................................. 230
TABLE B–8 Kerberos v5 Status Codes 4 .................................................................. 231
TABLE B–9 Kerberos v5 Status Codes 5 ................................................................. 232
TABLE B–10 Kerberos v5 Status Codes 6 ............................................................... 234
TABLE B–11 Kerberos v5 Status Codes 7 ............................................................... 235
TABLE E–1 SASL Functions Common to Clients and Servers ............................... 265
TABLE E–2 Basic SASL Client-only Functions ....................................................... 266
TABLE E–3 Basic SASL Server Functions (Clients Optional) ................................. 266
TABLE E–4 SASL Functions for Configuring Basic Services ................................. 266
TABLE E–5 SASL Utility Functions .......................................................................... 267
TABLE E–6 SASL Property Functions ...................................................................... 267
TABLE E–7 Callback Data Types ........................................................................... 268
TABLE E–8 SASL Include Files ............................................................................. 268
TABLE E–9 SASL Return Codes: General ............................................................... 268
TABLE E–10 SASL Return Codes: Client-Only ......................................................... 269
TABLE E–11 SASL Return Codes: Server-Only ....................................................... 269
TABLE E–12 SASL Return Codes – Password Operations ..................................... 270
Examples

EXAMPLE 2–1 Superuser Privilege Bracketing Example .......................................................... 30
EXAMPLE 2–2 Least Privilege Bracketing Example .............................................................. 31
EXAMPLE 2–3 Checking for Authorizations .............................................................................. 34
EXAMPLE 3–1 Sample PAM Consumer Application ............................................................... 42
EXAMPLE 3–2 PAM Conversation Function ............................................................................. 45
EXAMPLE 3–3 Sample PAM Service Module ........................................................................... 51
EXAMPLE 4–1 Using Strings in GSS-API ................................................................................. 60
EXAMPLE 4–2 Using gss_import_name() ................................................................................. 61
EXAMPLE 4–3 OIDs Structure ................................................................................................. 67
EXAMPLE 4–4 OID Set Structure ............................................................................................. 67
EXAMPLE 5–1 gss-client Example: main() ................................................................................ 95
EXAMPLE 5–2 connect_to_server() Function ......................................................................... 96
EXAMPLE 5–3 client_establish_context() – Translate Service Name ....................................... 98
EXAMPLE 5–4 Loop for Establishing Contexts ....................................................................... 100
EXAMPLE 5–5 gss-client: call_server() Establish Context .................................................... 102
EXAMPLE 5–6 gss-client Example: call_server() – Wrap Message ....................................... 103
EXAMPLE 5–7 gss-client Example – Read and Verify Signature Block .................................... 105
EXAMPLE 5–8 gss-client Example: call_server() – Delete Context ........................................ 106
EXAMPLE 6–1 gss-server Example: main() .............................................................................. 109
EXAMPLE 6–2 Sample Code for server_acquire_creds() Function ........................................ 112
EXAMPLE 6–3 sign_server() Function .................................................................................... 114
EXAMPLE 6–4 server_establish_context() Function ............................................................. 115
EXAMPLE 6–5 test_import_export_context() ......................................................................... 120
EXAMPLE 8–1 Supplying _fini() to PKCS #11 Libraries ......................................................... 152
EXAMPLE 9–1 Creating a Message Digest Using PKCS #11 Functions .................................. 163
EXAMPLE 9–2 Creating an Encryption Key Object Using PKCS #11 Functions ....................... 166
EXAMPLE 9–3 Signing and Verifying Text Using PKCS #11 Functions .................................. 170
EXAMPLE 9–4 Generating Random Numbers Using PKCS #11 Functions .......................... 177
<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE A-1</td>
<td>Complete Listing of gss-client.c Sample Program</td>
<td>187</td>
</tr>
<tr>
<td>EXAMPLE A-2</td>
<td>Complete Code Listing for gss-server.c Sample Program</td>
<td>197</td>
</tr>
<tr>
<td>EXAMPLE A-3</td>
<td>Code Listings for Miscellaneous GSS-API Functions</td>
<td>207</td>
</tr>
<tr>
<td>EXAMPLE B-1</td>
<td>Displaying Status Codes with gss_display_status()</td>
<td>220</td>
</tr>
<tr>
<td>EXAMPLE C-1</td>
<td>The /etc/gss/mech File</td>
<td>240</td>
</tr>
<tr>
<td>EXAMPLE C-2</td>
<td>The /etc/gss/qop File</td>
<td>240</td>
</tr>
<tr>
<td>EXAMPLE C-3</td>
<td>createMechOid() Function</td>
<td>242</td>
</tr>
<tr>
<td>EXAMPLE C-4</td>
<td>parse_oid() Function</td>
<td>243</td>
</tr>
</tbody>
</table>
Preface

The *Developer’s Guide to Oracle Solaris Security* describes the public application programming interfaces (API) and service provider interfaces (SPI) for the security features in the Oracle Solaris operating system. The term *service provider* refers to components that are plugged into a framework to provide security services, such as cryptographic algorithms and security protocols.

*Note* – This Solaris release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems appear in the *Solaris OS: Hardware Compatibility Lists* at [http://www.oracle.com/webfolder/technetwork/hcl/index.html](http://www.oracle.com/webfolder/technetwork/hcl/index.html). This document cites any implementation differences between the platform types.

In this document these x86 related terms mean the following:

- “x86” refers to the larger family of 64-bit and 32-bit x86 compatible products.
- “x64” points out specific 64-bit information about AMD64 or EM64T systems.
- “32-bit x86” points out specific 32-bit information about x86 based systems.

For supported systems, see the *Solaris OS: Hardware Compatibility Lists*.

Who Should Use This Book

The *Developer’s Guide to Oracle Solaris Security* is intended for C-language developers who want to write the following types of programs:

- Privileged applications that can override system controls
- Applications that use authentication and related security services
- Applications that need to secure network communications
- Applications that use cryptographic services
- Libraries, shared objects, and plug-ins that provide or consume security services
Preface

Note – For Java-language equivalents to the Oracle Solaris features, see http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.

Before You Read This Book

Readers of this guide should be familiar with C programming. A basic knowledge of security mechanisms is helpful but not required. You do not need to have specialized knowledge about network programming to use this book.

How This Book Is Organized

This book is organized into the following chapters.

- Chapter 1, “Oracle Solaris Security for Developers (Overview),” provides an introduction to Oracle Solaris security.
- Chapter 2, “Developing Privileged Applications,” describes how to write privileged applications that use process privileges.
- Chapter 3, “Writing PAM Applications and Services,” explains how to write a pluggable application module (PAM).
- Chapter 4, “Writing Applications That Use GSS-API,” provides an introduction to the Generic Security Service Application Programming Interface (GSS-API).
- Chapter 5, “GSS-API Client Example,” and Chapter 6, “GSS-API Server Example,” each provide a walk-through of GSS-API examples.
- Chapter 8, “Introduction to the Oracle Solaris Cryptographic Framework,” provides an overview of the Oracle Solaris cryptographic framework, both at the user level and kernel level.
- Chapter 9, “Writing User-Level Cryptographic Applications and Providers,” describes how to write consumers and providers for the user level of the Solaris cryptographic framework.
- Appendix A, “Sample C-Based GSS-API Programs,” provides complete code listings for the GSS-API examples.
- Appendix B, “GSS-API Reference,” provides reference information for various items in the GSS-API.
- Appendix C, “Specifying an OID,” describes how to specify a mechanism. This technique is used in cases where a mechanism other than the default mechanism is to be used.
Appendix D, “Source Code for SASL Example,” provides complete code listings for the SASL examples.

Appendix E, “SASL Reference Tables,” provides brief descriptions of the major SASL interfaces.

Appendix F, “Packaging and Signing Cryptographic Providers,” describes how to package and sign cryptographic providers.

Glossary provides definitions for security terms that are used throughout the manual.

Related Documentation

For other information about security features, see the following sources:

- System Administration Guide: Security Services provides descriptions of Oracle Solaris security features from the system administrator’s point of view.
- Application Packaging Developer’s Guide provides information about designing and building packages for Oracle Solaris 10 systems.
- ONC+ Developer’s Guide provides information about remote procedure calls.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.
**TABLE P–1  Typographic Conventions**

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, and onscreen computer output</td>
<td>Edit your .login file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use <code>ls -a</code> to list all files.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>machine_name%</code> you have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with onscreen computer output</td>
<td><code>machine_name% su</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Password:</code></td>
</tr>
<tr>
<td>aabbcc123</td>
<td>Placeholder: replace with a real name or value</td>
<td>The command to remove a file is <code>rm filename</code>.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new terms, and terms to be emphasized</td>
<td>Read Chapter 6 in the <em>User's Guide</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A <em>cache</em> is a copy that is stored locally.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do not save the file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Some emphasized items appear bold online.</td>
</tr>
</tbody>
</table>

**Shell Prompts in Command Examples**

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

**TABLE P–2  Shell Prompts**

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash shell, Korn shell, and Bourne shell</td>
<td>$</td>
</tr>
<tr>
<td>Bash shell, Korn shell, and Bourne shell for superuser</td>
<td>#</td>
</tr>
<tr>
<td>C shell</td>
<td><code>machine_name%</code></td>
</tr>
<tr>
<td>C shell for superuser</td>
<td><code>machine_name#</code></td>
</tr>
</tbody>
</table>
This manual documents the public application programming interfaces (APIs) and service provider interfaces (SPIs) for the security features in the Oracle Solaris Operating System (Oracle Solaris OS).

This chapter covers the following areas:
- “System Security” on page 19
- “Network Security Architecture” on page 21

**Overview of Oracle Solaris Security Features for Developers**

This manual covers the public APIs and public SPIs to security features in the Oracle Solaris operating system. For information about how these security features operate from the system administrator’s viewpoint, see Chapter 1, “Security Services (Overview),” in *System Administration Guide: Security Services*.

The Oracle Solaris OS provides a network security architecture that is based on standard industry interfaces. Through the use of standardized interfaces, applications that consume or provide cryptographic services should need no modification as security technologies evolve.

**System Security**

For system security, the Oracle Solaris OS provides process privileges. *Process privileges* are an alternative to the standard, superuser-based UNIX model for granting access to privileged applications. The system administrator assigns users a set of process privileges that permit access to privileged applications. A user does not need to become superuser to use a privileged application.

Privileges enable system administrators to delegate limited permission to users to override system security instead of giving users complete root access. Accordingly, developers who...
create new privileged applications should test for specific privileges instead of checking for UID = 0. See Chapter 2, “Developing Privileged Applications.”

For highly stringent system security, consider using the Oracle Solaris’ Trusted Extensions feature. The Trusted Extensions feature enables system administrators to specify the applications and files that a particular user can access. The Trusted Extensions feature is outside of the scope of this book. See http://www.oracle.com/us/sun/index.htm for more information.

The Oracle Solaris OS provides the following public interfaces for security:

- **Cryptographic framework** – The cryptographic framework is the backbone of cryptographic services in the Oracle Solaris OS. The framework provides standard PKCS #11 interfaces to accommodate consumers and providers of cryptographic services. The framework has two parts: the user cryptographic framework for user-level applications and the kernel cryptographic framework for kernel-level modules. Consumers that are connected to the framework need no special knowledge of the installed cryptographic mechanisms. Providers plug into the framework with no special code necessary for the different types of consumers.

  The consumers of the cryptographic framework include security protocols, certain mechanisms, and applications that need to perform cryptography. The providers to the framework are cryptographic mechanisms as well as other mechanisms in hardware and software plug-ins. See Chapter 8, “Introduction to the Oracle Solaris Cryptographic Framework,” for an overview of the cryptographic framework. See Chapter 9, “Writing User-Level Cryptographic Applications and Providers,” to learn how to write user-level applications that consume services from the framework.

  The library for the cryptographic framework is an implementation of the RSA PKCS#11 v2.11 specification. Both consumers and providers communicate with the user-level cryptographic framework through standard PKCS #11 calls.

- **Java API** – Java security technology includes a large set of APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols. The Java security APIs span a wide range of areas, including cryptography, public key infrastructure, secure communication, authentication, and access control. Java security technology provides the developer with a comprehensive security framework for writing applications, and also provides the user or administrator with a set of tools to securely manage applications. See http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.
Network Security Architecture

The network security architecture works with standard industry interfaces, such as PAM, GSS-API, SASL, and RSA Security Inc. PKCS#11 Cryptographic Token Interface (Cryptoki). Through the use of standardized protocols and interfaces, developers can write both consumers and providers that need no modification as security technologies evolve.

An application, library, or kernel module that uses security services is called a consumer. An application that provides security services to consumers is referred to as a provider and also as a plug-in. The software that implements a cryptographic operation is called a mechanism. A mechanism is not just an algorithm but includes the manner in which the algorithm is to be applied. For example, one mechanism might apply the DES algorithm to authentication. A different mechanism might apply DES to data protection with block-by-block encryption.

The network security architecture eliminates the need for developers of consumers to write, maintain, and optimize cryptographic algorithms. Optimized cryptographic mechanisms are provided as part of the architecture.

The Oracle Solaris OS provides the following public interfaces for security:

- **PAM** – Pluggable authentication modules. PAM modules are mainly used for the initial authentication of a user to a system. The user can enter the system by GUI, command line, or some other means. In addition to authentication services, PAM provides services for managing accounts, sessions, and passwords. Applications such as `login`, `rlogin`, and `telnet` are typical consumers of PAM services. The PAM SPI is supplied services by security providers such as Kerberos v5 and Smartcard. See Chapter 3, "Writing PAM Applications and Services."

- **GSS-API** – Generic security service application program interface. The GSS-API provides secure communication between peer applications. The GSS-API provides authentication, integrity, and confidentiality protection services as well. The Solaris implementation of the GSS-API works with Kerberos v5, SPNEGO, and Diffie-Hellman encryption. The GSS-API is primarily used to design or implement secure application protocols. GSS-API can provide services to other kinds of protocols, such as SASL. Through SASL, GSS-API provides services to LDAP.

GSS-API is typically used by two peer applications that are communicating over a network after the initial establishment of credentials has occurred. GSS-API is used by login applications, NFS, and ftp, among other applications.

See Chapter 4, "Writing Applications That Use GSS-API," for an introduction to GSS-API. Chapter 5, "GSS-API Client Example," and Chapter 6, "GSS-API Server Example," provides the source code descriptions of two typical GSS-API applications. Appendix A, "Sample C-Based GSS-API Programs," presents the code listings for the GSS-API examples. Appendix B, "GSS-API Reference," provides reference material for GSS-API. Appendix C, "Specifying an OID," demonstrates how to specify a mechanism other than the default mechanism.
- **SASL** – Simple authentication and security layer. SASL is used largely by protocols, for authentication, privacy, and data integrity. SASL is intended for higher-level network-based applications that use dynamic negotiation of security mechanisms to protect sessions. LDAP is one of the better-known consumers of SASL. SASL is similar to GSS-API. SASL is on a somewhat higher level than GSS-API. SASL consumes GSS-API services. See Chapter 7, “Writing Applications That Use SASL.”

- **Smart Card** – Developers of IFD handlers for smart card terminals can provide services to consumers through the terminal interface to the smartcard framework. Information on these interfaces is provided in Chapter 10, “Using the Smart Card Framework.”
This chapter describes how to develop privileged applications. The chapter covers the following topics:

- “Privileged Applications” on page 23
- “About Privileges” on page 24
- “Programming with Privileges” on page 27
- “About Authorizations” on page 34

Privileged Applications

A privileged application is an application that can override system controls and check for specific user IDs (UIDs), group IDs (GIDs), authorizations, or privileges. These access control elements are assigned by system administrators. For a general discussion of how administrators use these access control elements, see Chapter 8, “Using Roles and Privileges (Overview),” in System Administration Guide: Security Services.

The Oracle Solaris OS provides developers with two elements that enable a finer-grained delegation of privileges:

- **Privileges** – A privilege is a discrete right that can be granted to an application. With a privilege, a process can perform an operation that would otherwise be prohibited by the Oracle Solaris OS. For example, processes cannot normally open data files without the proper file permission. The `file_dac_read` privilege provides a process with the ability to override the UNIX file permissions for reading a file. Privileges are enforced at the kernel level.

- **Authorizations** – An authorization is a permission for performing a class of actions that are otherwise prohibited by security policy. An authorization can be assigned to a role or user. Authorizations are enforced at the user level.

The difference between authorizations and privileges has to do with the level at which the policy of who can do what is enforced. Privileges are enforced at the kernel level. Without the proper
privilege, a process cannot perform specific operations in a privileged application. Authorizations enforce policy at the user application level. An authorization might be required for access to a privileged application or for specific operations within a privileged application.

**About Privileges**

A privilege is a discrete right that is granted to a process to perform an operation that would otherwise be prohibited by the Oracle Solaris OS. Most programs do not use privileges, because a program typically operates within the bounds of the system security policy.

Privileges are assigned by an administrator. Privileges are enabled according to the design of the program. At login or when a profile shell is entered, the administrator’s privilege assignments apply to any commands that are executed in the shell. When an application is run, privileges are turned on or turned off programmatically. If a new program is started by using the `exec(1)` command, that program can potentially use all of the parent process’s inheritable privileges. However, that program cannot add any new privileges.

**How Administrators Assign Privileges**

System administrators are responsible for assigning privileges to commands. For more information about privilege assignment, see “Privileges (Overview)” in *System Administration Guide: Security Services*.

**How Privileges Are Implemented**

Every process has four sets of privileges that determine whether a process can use a particular privilege:

- Permitted privilege set
- Inheritable privilege set
- Limit privilege set
- Effective privilege set

**Permitted Privilege Set**

All privileges that a process can ever potentially use must be included in the permitted set. Conversely, any privilege that is never to be used should be excluded from the permitted set for that program.

When a process is started, that process inherits the permitted privilege set from the parent process. Typically at login or from a new profile shell, all privileges are included in the initial set of permitted privileges. The privileges in this set are specified by the administrator. Each child
process can remove privileges from the permitted set, but the child cannot add other privileges to the permitted set. As a security precaution, you should remove those privileges from the permitted set that the program never uses. In this way, a program can be protected from using an incorrectly assigned or inherited privilege.

Privileges that are removed from the permitted set are automatically removed from the effective set.

**Inheritable Privilege Set**

At login or from a new profile shell, the inheritable set contains the privileges that have been specified by the administrator. These inheritable privileges can potentially be passed on to child processes after an `exec(1)` call. A process should remove any unnecessary privileges to prevent these privileges from passing on to a child process. Often the permitted and inheritable sets are the same. However, there can be cases where a privilege is taken out of the inheritable set, but that privilege remains in the permitted set.

**Limit Privilege Set**

The limit set enables a developer to control which privileges a process can exercise or pass on to child processes. A child process and the descendant processes can only obtain privileges that are in the limit set. When a `setuid(0)` function is executed, the limit set determines the privileges that the application is permitted to use. The limit set is enforced at `exec(1)` time. Removal of privileges from the limit set does not affect any other sets until the `exec(1)` is performed.

**Effective Privilege Set**

The privileges that a process can actually use are in the process's effective set. At the start of a program, the effective set is equal to the permitted set. Afterwards, the effective set is either a subset of or is equal to the permitted set.

A good practice is to reduce the effective set to the set of basic privileges. The basic privilege set, which contains the core privileges, is described in "Privilege Categories" on page 26. Remove completely any privileges that are not needed in the program. Toggle off any basic privileges until that privilege is needed. For example, the `file_dac_read` privilege, enables all files to be read. A program can have multiple routines for reading files. The program turns off all privileges initially and turns on `file_dac_read`, for appropriate reading routines. The developer thus ensures that the program cannot exercise the `file_dac_read` privilege for the wrong reading routines. This practice is called privilege bracketing. Privilege bracketing is demonstrated in "Privilege Coding Example" on page 30.
Compatibility Between the Superuser and Privilege Models

To accommodate legacy applications, the implementation of privileges works with both the superuser and privilege models. This accommodation is achieved through use of the PRIV_AWARE flag, which indicates that a program works with privileges. The PRIV_AWARE flag is handled automatically by the operating system.

Consider a child process that is not aware of privileges. The PRIV_AWARE flag for that process would be false. Any privileges that have been inherited from the parent process are available in the permitted and effective sets. If the child sets a UID to 0, the process’s effective and permitted sets are restricted to those privileges in the limit set. The child process does not gain full superuser powers. Thus, the limit set of a privilege-aware process restricts the superuser privileges of any non-privilege-aware child processes. If the child process modifies any privilege set, then the PRIV_AWARE flag is set to true.

Privilege Categories

Privileges are logically grouped on the basis of the scope of the privilege, as follows:

- Basic privileges – The core privileges that are needed for minimal operation. The basic privileges are as follows:
  - PRIV_FILE_LINK_ANY – Allows a process to create hard links to files that are owned by a UID other than the process’s effective UID.
  - PRIV_PROC_EXEC – Allows a process to call execve().
  - PRIV_PROC_FORK – Allows a process to call fork(), fork1(), or vfork().
  - PRIV_PROC_SESSION – Allows a process to send signals or trace processes outside its session.
  - PRIV_PROC_INFO – Allows a process to examine the status of processes outside of those processes to which the inquiring process can send signals. Without this privilege, processes that cannot be seen in /proc cannot be examined.

In general, the basic privileges should be assigned as a set rather than individually. This approach ensures that any basic privileges that are released in an update to the Oracle Solaris OS will be included in the assignment. On the other hand, a privilege that is known not to be needed by a program should be explicitly turned off. For example, the proc_exec privilege should be turned off if the program is not intended to exec(1) sub-processes.

- File system privileges.
- System V Interprocess Communication (IPC) privileges.
- Network privileges.
- Process privileges.
System privileges.

See the `privileges(5)` man page for a complete list of the Solaris privileges with descriptions.

**Note** - Solaris provides the zones facility, which lets an administrator set up isolated environments for running applications. See `zones(5)`. Since a process in a zone is prevented from monitoring or interfering with other activity in the system outside of that zone, any privileges on that process are limited to the zone as well. However, if needed, the PRIV_PROC_ZONE privilege can be applied to processes in the global zone that need privileges to operate in non-global zones.

**Programming with Privileges**

This section discusses the interfaces for working with privileges. To use the privilege programming interfaces, you need the following header file.

```c
#include <priv.h>
```

An example demonstrating how privilege interfaces are used in a privileged application is also provided.

**Privilege Data Types**

The major data types that are used by the privilege interfaces are:

- **Privilege type** – An individual privilege is represented by the `priv_t` type definition. You initialize a variable of type `priv_t` with a privilege ID string, as follows:

  ```c
  priv_t priv_id = PRIV_FILE_DAC_WRITE;
  ```

- **Privilege set type** – Privilege sets are represented by the `priv_set_t` data structure. Use one of the privilege manipulation functions shown in Table 2-1 to initialize variables of type `priv_set_t`.

- **Privilege operation type** – The type of operation to be performed on a file or process privilege set is represented by the `priv_op_t` type definition. Not all operations are valid for every type of privilege set. Read the privilege set descriptions in “Programming with Privileges” on page 27 for details.

  Privilege operations can have the following values:

  - **PRIV_ON** – Turn the privileges that have been asserted in the `priv_set_t` structure on in the specified file or process privilege set.
  - **PRIV_OFF** – Turn the privileges asserted in the `priv_set_t` structure off in the specified file or process privilege set.
- **PRIV_SET** – Set the privileges in the specified file or process privilege set to the privileges asserted in the `priv_set_t` structure. If the structure is initialized to empty, PRIV_SET sets the privilege set to none.

### Privilege Interfaces

The following table lists the interfaces for using privileges. Descriptions of some major privilege interfaces are provided after the table.

<table>
<thead>
<tr>
<th>TABLE 2–1 Interfaces for Using Privileges</th>
<th>Purpose</th>
<th>Functions</th>
<th>Additional Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting and setting privilege sets</td>
<td></td>
<td><code>setppriv(2), getppriv(2), priv_set(3C), priv_ineffect(3C)</code></td>
<td><code>setppriv()</code> and <code>getppriv()</code> are system calls. <code>priv_ineffect()</code> and <code>priv_set()</code> are wrappers for convenience.</td>
</tr>
<tr>
<td>Identifying and translating privileges</td>
<td></td>
<td><code>priv_str_to_set(3C), priv_set_to_str(3C), priv_getbyname(3C), priv_getbynum(3C), priv_getsetbyname(3C), priv_getsetbynum(3C)</code></td>
<td>These functions map the specified privilege or privilege set to a name or a number.</td>
</tr>
<tr>
<td>Manipulating privilege sets</td>
<td></td>
<td><code>priv_allocset(3C), priv_freeset(3C), priv_emptyset(3C), priv_fillset(3C), priv_isemptyset(3C), priv_isfullset(3C), priv_isequalset(3C), priv_issubset(3C), priv_intersect(3C), priv_union(3C), priv_inverse(3C), priv_addset(3C), priv_copyset(3C), priv_delset(3C), priv_ismember(3C)</code></td>
<td>These functions are concerned with privilege memory allocation, testing, and various set operations.</td>
</tr>
<tr>
<td>Getting and setting process flags</td>
<td></td>
<td><code>getpflags(2), setpflags(2)</code></td>
<td>The PRIV_AWARE process flag indicates whether the process understands privileges or runs under the superuser model. PRIV_DEBUG is used for privilege debugging.</td>
</tr>
</tbody>
</table>
setppriv() for Setting Privileges

The main function for setting privileges is `setppriv()`, which has the following syntax:

```c
int setppriv(priv_op_t op, priv_ptype_t which, const priv_set_t *set);
```

`op` represents the privilege operation that is to be performed. The `op` parameter has one of three possible values:

- **PRIV_ON** – Adds the privileges that are specified by the `set` variable to the set type that is specified by `which`.
- **PRIV_OFF** – Removes the privileges that are specified by the `set` variable from the set type that is specified by `which`.
- **PRIV_SET** – Uses the privileges that are specified by the `set` variable to replace privileges in the set type that is specified by `which`.

`which` specifies the type of privilege set to be changed, as follows:

- **PRIV_PERMITTED**
- **PRIV_EFFECTIVE**
- **PRIV_INHERITABLE**
- **PRIV_LIMIT**

`set` specifies the privileges to be used in the change operation.

In addition, a convenience function is provided: `priv_set()`.

`priv_str_to_set()` for Mapping Privileges

These functions are convenient for mapping privilege names with their numeric values. `priv_str_to_set()` is a typical function in this family. `priv_str_to_set()` has the following syntax:

```c
priv_set_t *priv_str_to_set(const char *buf, const char *set, const char **endptr);
```

`priv_str_to_set()` takes a string of privilege names that are specified in `buf`. `priv_str_to_set()` returns a set of privilege values that can be combined with one of the four privilege sets. `**endptr` can be used to debug parsing errors. Note that the following keywords can be included in `buf`:
“all” indicates all defined privileges. “all, !priv_name, . . .” enables you to specify all privileges except the indicated privileges.

Note — Constructions that use “priv_set, “!priv_name, . . .” subtract the specified privilege from the specified set of privileges. Do not use “!priv_name, . . .” without first specifying a set because with no privilege set to subtract from, the construction subtracts the specified privileges from an empty set of privileges and effectively indicates no privileges.

“none” indicates no privileges.
“basic” indicates the set of privileges that are required to perform operations that are traditionally granted to all users on login to a standard UNIX operating system.

Privilege Coding Example

This section compares how privileges are bracketed using the superuser model and the least privilege model.

Privilege Bracketing in the Superuser Model

The following example demonstrates how privileged operations are bracketed in the superuser model.

EXAMPLE 2–1  Superuser Privilege Bracketing Example
/* Program start */
uid = getuid();
seteuid(uid);

/* Privilege bracketing */
seteuid(0);
/* Code requiring superuser capability */
...
/* End of code requiring superuser capability */
seteuid(uid);
...
/* Give up superuser ability permanently */
setreuid(uid,uid);

Privilege Bracketing in the Least Privilege Model

This example demonstrates how privileged operations are bracketed in the least privilege model. The example uses the following assumptions:

- The program is setuid 0.
- The permitted and effective sets are initially set to all privileges as a result of setuid 0.
- The inheritable set is initially set to the basic privileges.
The limit set is initially set to all privileges.

An explanation of the example follows the code listing.

**EXAMPLE 2-2 Least Privilege Bracketing Example**

```c
#include <priv.h>
/* Always use the basic set. The Basic set might grow in future
 * releases and potentially restrict actions that are currently
 * unrestricted */
priv_set_t *temp = priv_str_to_set("basic", ",", NULL);

/* PRIV_FILE_DAC_READ is needed in this example */
(void) priv_addset(temp, PRIV_FILE_DAC_READ);

/* PRIV_PROC_EXEC is no longer needed after program starts */
(void) priv_delset(temp, PRIV_PROC_EXEC);

/* Compute the set of privileges that are never needed */
priv_inverse(temp);

/* Remove the set of unneeded privs from Permitted (and by
 * implication from Effective) */
(void) setppriv(PRIV_OFF, PRIV_PERMITTED, temp);

/* Remove unneeded priv set from Limit to be safe */
(void) setppriv(PRIV_OFF, PRIV_LIMIT, temp);

/* Done with temp */
priv_freeset(temp);

/* Now get rid of the euid that brought us extra privs */
(void) seteuid(getuid());

/* Toggle PRIV_FILE_DAC_READ off while it is unneeded */
priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_DAC_READ, NULL);

/* Toggle PRIV_FILE_DAC_READ on when special privilege is needed*/
priv_set(PRIV_ON, PRIV_EFFECTIVE, PRIV_FILE_DAC_READ, NULL);

fd = open("/some/restricted/file", O_RDONLY);

/* Toggle PRIV_FILE_DAC_READ off after it has been used */
priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_DAC_READ, NULL);

/* Remove PRIV_FILE_DAC_READ when it is no longer needed */
priv_set(PRIV_OFF, PRIV_ALLSETS, PRIV_FILE_DAC_READ, NULL);
```

The program defines a variable that is named `temp`. The `temp` variable determines the set of privileges that are not needed by this program. Initially in line 5, `temp` is defined to contain the set of basic privileges. In line 7, the `file_dac_read` privilege is added to `temp`. The `proc_exec` privilege is necessary to `exec(1)` new processes, which is not permitted in this program. Therefore, `proc_exec` is removed from `temp` in line 9 so that the `exec(1)` command cannot execute new processes.
At this point, temp contains only those privileges that are needed by the program, that is, the basic set plus file_dac_read minus proc_exec. In line 11, the priv_inverse() function computes the inverse of temp and resets the value of temp to the inverse. The inverse is the result of subtracting the specified set, temp in this case, from the set of all possible privileges. As a result of line 11, temp now contains those privileges that are never needed by the program. In line 14, the unneeded privileges that are defined by temp are subtracted from the permitted set. This removal effectively removes the privileges from the effective set as well. In line 16, the unneeded privileges are removed from the limit set. In line 18, the temp variable is freed, since temp is no longer needed.

This program is aware of privileges. Accordingly, the program does not use setuid and can reset the effective UID to the user's real UID in line 20.

The file_dac_read privilege is turned off in line 22 through removal from the effective set. In a real program, other activities would take place before file_dac_read is needed. In this sample program, file_dac_read is needed for to read a file in line 25. Accordingly, file_dac_read is turned on in line 24. Immediately after the file is read, file_dac_read is again removed from the effective set. When all files have been read, file_dac_read is removed for good by turning off file_dac_read in all privilege sets.

The following table shows the transition of the privilege sets as the program progresses. The line numbers are indicated.

<table>
<thead>
<tr>
<th>Step</th>
<th>temp Set</th>
<th>Permitted Privilege Set</th>
<th>Effective Privilege Set</th>
<th>Limit Privilege Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>–</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 5 – temp is set to basic privileges</td>
<td>basic</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 7 – file_dac_read is added to temp.</td>
<td>basic + file_dac_read</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 9 – proc_exec is removed from temp.</td>
<td>basic + file_dac_read - proc_exec</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 11 – temp is reset to the inverse.</td>
<td>all – (basic + file_dac_read - proc_exec)</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 14 – The unneeded privileges are turned off in the permitted set.</td>
<td>all – (basic + file_dac_read - proc_exec)</td>
<td>basic + file_dac_read - proc_exec</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Line 16 – The unneeded privileges are turned off in the limit set.</td>
<td>all – (basic + file_dac_read - proc_exec)</td>
<td>basic + file_dac_read - proc_exec</td>
<td>basic + file_dac_read - proc_exec</td>
<td>all</td>
</tr>
</tbody>
</table>
Guidelines for Developing Privileged Applications

This section provides the following suggestions for developing privileged applications:

- **Use an isolated system.** You should never debug privileged applications on a production system, as an incomplete privileged application can compromise security.

- **Set IDs properly.** The calling process needs the `proc_setid` privilege in its effective set to change its user ID, group ID, or supplemental group ID.

- **Use privilege bracketing.** When an application uses privilege, system security policy is being overridden. Privileged tasks should be bracketed and carefully controlled to ensure that sensitive information is not compromised. See "Privilege Coding Example" on page 30 for information about how to bracket privileges.

- **Start with the basic privileges.** The basic privileges are necessary for minimal operation. A privileged application should start with the basic set. The application should then subtract and add privileges appropriately. A typical start-up scenario follows.

1. The daemon starts up as root.
2. The daemon turns on the basic privilege set.
3. The daemon turns off any basic privileges that are unnecessary, for example, PRIV_FILE_LINK_ANY.
4. The daemon adds any other privileges that are needed, for example, PRIV_FILE_DAC_READ.
5. The daemon switches to the daemon UID.
Avoid shell escapes. The new process in a shell escape can use any of the privileges in the parent process's inheritable set. An end user can therefore potentially violate trust through a shell escape. For example, some mail applications might interpret the `!` command line as a command and would execute that line. An end user could thus create a script to take advantage of any mail application privileges. The removal of unnecessary shell escapes is a good practice.

About Authorizations

Authorizations are stored in the `/etc/security/auth_attr` file. To create an application that uses authorizations, take the following steps:

1. Scan the `/etc/security/auth_attr` for one or more appropriate authorizations.
2. Check for the required authorization at the beginning of the program using the `chkauthattr(3SECDB)` function. The `chkauthattr()` function searches for the authorization in order in the following locations:
   - AUTHS_GRANTED key in the `policy.conf(4)` database – AUTHS_GRANTED indicates authorizations that have been assigned by default.
   - PROFS_GRANTED key in the `policy.conf(4)` database – PROFS_GRANTED indicates rights profiles that have been assigned by default. `chkauthattr()` checks these rights profiles for the specified authorization.
   - The `user_attr(4)` database – This database stores security attributes that have been assigned to users.
   - The `prof_attr(4)` database – This database stores rights profiles that have been assigned to users.
   
   If `chkauthattr()` cannot find the right authorization in any of these places, then the user is denied access to the program.
3. Let the administrator know which authorizations are required for this application. You can inform the administrators through man pages or other documentation.

EXAMPLE 2–3  Checking for Authorizations

The following code snippet demonstrates how the `chkauthattr()` function can be used to check a user’s authorization. In this case, the program checks for the `solaris.job.admin` authorization. If the user has this authorization, the user is able to read or write to other users’ files. Without the authorization, the user can operate on owned files only.

```c
/* Define override privileges */
priv_set_t *override_prv = priv_allocset();

/* Clear privilege set before adding privileges. */
priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_DAC_READ,
```
priv_FILE_DAC_WRITE, NULL);

priv_addset(override_prv, PRIV_FILE_DAC_READ);
priv_addset(override_prv, PRIV_FILE_DAC_WRITE);

if (!chkauthattr("solaris.jobs.admin", username)) {
    /* turn off privileges */
    setppriv(PRIV_OFF, PRIV_EFFECTIVE, override_prv);
}

/* Authorized users continue to run with privileges */
/* Other users can read or write to their own files only */
Writing PAM Applications and Services

Pluggable authentication modules (PAM) provide system entry applications with authentication and related security services. This chapter is intended for developers of system entry applications who wish to provide authentication, account management, session management, and password management through PAM modules. There is also information for designers of PAM service modules. The following topics are discussed:

- "Introduction to the PAM Framework" on page 37
- "PAM Configuration" on page 40
- "Writing Applications That Use PAM Services" on page 41
- "Writing Modules That Provide PAM Services" on page 49

PAM was originally developed at Oracle Corporation. The PAM specification has since been submitted to X/Open, which is now the Open Group. The PAM specification is available in *X/Open Single Sign-On Service (XSSO) - Pluggable Authentication*, Open Group, UK ISBN 1-85912-144-6 June 1997. The Oracle Solaris implementation of PAM is described in the *pam(3PAM)*, *libpam(3LIB)*, and *pam_sm(3PAM)* man pages.

Introduction to the PAM Framework

The PAM framework consists of four parts:

- PAM consumers
- PAM library
- The *pam.conf(4)* configuration file
- PAM service modules, also referred to as providers

The framework provides a uniform way for authentication-related activities to take place. This approach enables application developers to use PAM services without having to know the semantics of the policy. Algorithms are centrally supplied. The algorithms can be modified independently of the individual applications. With PAM, administrators can tailor the authentication process to the needs of a particular system without having to change any applications. Adjustments are made through *pam.conf*, the PAM configuration file.
The following figure illustrates the PAM architecture. Applications communicate with the PAM library through the PAM application programming interface (API). PAM modules communicate with the PAM library through the PAM service provider interface (SPI). Thus, the PAM library enables applications and modules to communicate with each other.

**FIGURE 3-1** PAM Architecture

**PAM Service Modules**

A PAM service module is a shared library that provides authentication and other security services to system entry applications such as `login`, `rlogin`, and `telnet`. The four types of PAM services are:

- **Authentication service modules** – For granting users access to an account or service. Modules that provide this service authenticate users and set up user credentials.

- **Account management modules** – For determining whether the current user’s account is valid. Modules that provide this service can check password or account expiration and time-restricted access.

- **Session management modules** – For setting up and terminating login sessions.

- **Password management modules** – For enforcing password strength rules and performing authentication token updates.
A PAM module can implement one or more of these services. The use of simple modules with well-defined tasks increases configuration flexibility. PAM services should thus be implemented in separate modules. The services can then be used as needed as defined in the `pam.conf` file.

For example, the Oracle Solaris OS provides the `pam_authtok_check` module for system administrators to configure the site’s password policy. The `pam_authtok_check` module checks proposed passwords for various strength criteria.

For a complete list of Oracle Solaris PAM modules, see *man pages section 5: Standards, Environments, and Macros*. The PAM modules have the prefix `pam_`.

### PAM Library

The PAM library, `libpam`, is the central element in the PAM architecture:

- `libpam` exports an API, `pam`. Applications can call this API for authentication, account management, credential establishment, session management, and password changes.
- `libpam` imports a master configuration file `pam.conf`. The PAM configuration file specifies the PAM module requirements for each available service. `pam.conf` is managed by a system administrator.
- `libpam` imports an SPI, `pam_sm`, which is exported by the service modules.

### PAM Authentication Process

As an example of how consumers use the PAM library for user authentication, consider how `login` authenticates a user:

1. The `login` application initiates a PAM session by calling `pam_start` and by specifying the `login` service.
2. The application calls `pam_authenticate`, which is part of the PAM API that is exported by the PAM library, `libpam`.
3. The library searches for `login` entries in the `pam.conf` file.
4. For each module in `pam.conf` that is configured for the `login` service, the PAM library calls `pam_sm_authenticate`. The `pam_sm_authenticate()` function is part of the PAM SPI. The `pam.conf` control flag and results of each call determine whether the user is allowed access to the system. This process is described in more detail in *PAM Configuration (Reference)* in *System Administration Guide: Security Services*.

In this way, the PAM library connects PAM applications with the PAM modules that have been configured by the system administrator.
Requirements for PAM Consumers

PAM consumers must be linked with the PAM library libpam. Before an application can use any service that is provided by the modules, the application must initialize its instance of the PAM library by calling `pam_start(3PAM)`. The call to `pam_start()` initializes a handle that must be passed to all subsequent PAM calls. When an application is finished with the PAM services, `pam_end()` is called to clean up any data that was used by the PAM library.

Communication between the PAM application and the PAM modules takes place through items. For example, the following items are useful for initialization:

- PAM_USER – Currently authenticated user
- PAM_AUTHTOK – Password
- PAM_USER_PROMPT – User name prompt
- PAM_TTY – Terminal through which the user communication takes place
- PAM_RHOST – Remote host through which user enters the system
- PAM_REPOSITORY – Any restrictions on the user account repository
- PAM_RESOURCE – Any controls on resources

For a complete list of available items, see `pam_set_item(3PAM)`. Items can be set by the application through `pam_set_item(3PAM)`. Values that have been set by the modules can be retrieved by the application through `pam_get_item(3PAM)`. However, PAM_AUTHTOK and PAM_OLDAUTHHTOK cannot be retrieved by the application. The PAM_SERVICE item cannot be set.

---

Note – PAM consumers must have unique PAM service names which are passed to `pam_start(3PAM)`.

---

PAM Configuration

The PAM configuration file, `pam.conf(4)`, is used to configure PAM service modules for system services, such as `login`, `rlogin`, `su`, and `cron`. The system administrator manages this file. An incorrect order of entries in `pam.conf` can cause unforeseen side effects. For example, a badly configured `pam.conf` can lock out users so that single-user mode becomes necessary for repair. For information about PAM configuration, see “PAM Configuration (Reference)” in *System Administration Guide: Security Services*. 
This section provides a sample application that uses several PAM functions.

A Simple PAM Consumer Example

The following PAM consumer application is provided as an example. The example is a basic terminal-lock application that validates a user trying to access a terminal. The example goes through the following steps:

1. Initialize the PAM session.
   PAM sessions are initiated by calling the `pam_start(3PAM)` function. A PAM consumer application must first establish a PAM session before calling any of the other PAM functions. The `pam_start(3PAM)` function takes the following arguments:
   - `plock` – Service name, that is, the name of the application. The service name is used by the PAM framework to determine which rules in the configuration file, `/etc/pam.conf`, are applicable. The service name is generally used for logging and error-reporting.
   - `pw->pw_name` – The username is the name of the user that the PAM framework acts on.
   - `&conv` – The conversation function, `conv`, which provides a generic means for PAM to communicate with a user or application. Conversation functions are necessary because the PAM modules have no way of knowing how communication is to be conducted. Communication can be by means of GUIs, the command line, a smart card reader, or other devices. For more information, see “Writing Conversation Functions” on page 45.
   - `&pamh` – The PAM handle, `pamh`, which is an opaque handle that is used by the PAM framework to store information about the current operation. This handle is returned by a successful call to `pam_start()`.

   **Note** – An application that calls PAM interfaces must be sufficiently privileged to perform any needed operations such as authentication, password change, process credential manipulation, or audit state initialization. In this example, the application must be able to read `/etc/shadow` to verify the passwords for local users.

2. Authenticate the user.
   The application calls `pam_authenticate(3PAM)` to authenticate the current user. Generally, the user is required to enter a password or other authentication token depending on the type of authentication service. The PAM framework invokes the modules listed for the authentication service, `auth`, in `/etc/pam.conf`. The service name `plock` is used to determine which `pam.conf` entries are to be used. If there are no entries for `plock`, then the entries in other are used by default. If NULL passwords are explicitly disallowed in the application configuration file, then the `PAM_DISALLOW_NULL_AUTHTOK` flag should be passed. Solaris applications check the `PASSREQ=YES` setting in `/etc/default/login`. 
3. Check account validity.
   The example uses the `pam_acct_mgmt(3PAM)` function to check the validity of the authenticated user's account. In this example, `pam_acct_mgmt()` checks for expiration of the password.
   The `pam_acct_mgmt()` function also uses the PAM_DISALLOW_NULL_AUTHTOK flag. If `pam_acct_mgmt()` returns PAM_NEW_AUTHTOK_REQD, then `pam_chauthtok(3PAM)` should be called to allow the authenticated user to change the password.

4. Force the user to change passwords if the system discovers that the password has expired.
   The example uses a loop to call `pam_chauthtok()` until success is returned. The `pam_chauthtok()` function returns success if the user successfully changes his or her authentication information, which is usually the password. In this example, the loop continues until success is returned. More commonly, an application would set a maximum number of tries before terminating.

5. Call `pam_setcred(3PAM)`.
   The `pam_setcred(3PAM)` function is used to establish, modify, or delete user credentials. `pam_setcred()` is typically called when a user has been authenticated. The call is made after the account has been validated, but before a session has been opened. The `pam_setcred()` function is used with the PAM_ESTABLISH_CRED flag to establish a new user session. If the session is the renewal of an existing session, such as for lockscreens, `pam_setcred()` with the PAM_REFRESH_CRED flag should be called. If the session is changing the credentials, such as using `su` or assuming a role, then `pam_setcred()` with the PAM_REINITIALIZE_CRED flag should be called.

6. Close the PAM session.
   The PAM session is closed by calling the `pam_end(3PAM)` function. `pam_end()` frees all PAM resources as well.

The following example shows the source code for the sample PAM consumer application.

### EXAMPLE 3-1  Sample PAM Consumer Application

/*
 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <strings.h>
#include <signal.h>
#include <pwd.h>
#include <errno.h>
#include <security/pam_appl.h>

extern int pam_tty_conv(int num_msg, struct pam_message **msg,
/* Disable keyboard interrupts (Ctrl-C, Ctrl-Z, Ctrl-\) */
static void
disable_kbd_signals(void)
{
 (void) signal(SIGINT, SIG_IGN);
 (void) signal(SIGTSTP, SIG_IGN);
 (void) signal(SIGQUIT, SIG_IGN);
}

/* Terminate current user session, i.e., logout */
static void
logout()
{
 pid_t pgroup = getpgrp();

 (void) signal(SIGTERM, SIG_IGN);
 (void) fprintf(stderr, "Sorry, your session can't be restored.\n");
 (void) getchar();
 (void) kill(-pgroup, SIGTERM);
 (void) sleep(2);
 (void) kill(-pgroup, SIGKILL);
 exit(-1);
}

int /*ARGSUSED*/
main(int argc, char *argv)
{
 struct pam_conv conv = { pam_tty_conv, NULL };
 pam_handle_t *pamh;
 struct passwd *pw;
 int err;

disable_kbd_signals();
 if ((pw = getpwuid(getuid())) == NULL) {
 (void) fprintf(stderr, "plock: Can't get username: %s\n", 
 strerror(errno));
 exit(1);
 }

 /* Initialize PAM framework */
 err = pam_start("plock", pw->pw_name, &conv, &pamh);
 if (err != PAM_SUCCESS) {
 (void) fprintf(stderr, "plock: pam_start failed: %s\n", 
 pam_strerror(pamh, err));
 exit(1);
 }

 /* Authenticate user in order to unlock screen */
 do {
 (void) fprintf(stderr, "Terminal locked for %s. ", pw->pw_name);
 err = pam_authenticate(pamh, UNKNOWN);
 if (err == PAM_USER_UNKNOWN) {

EXAMPLE 3–1  Sample PAM Consumer Application  (Continued)

    logout();
    } else if (err != PAM_SUCCESS) {
        (void) fprintf(stderr, "Invalid password.
    }
    } while (err != PAM_SUCCESS);

    /* Make sure account and password are still valid */
    switch (err = pam_acct_mgmt(pamh, 0)) {
    case PAM_SUCCESS:
        break;
    case PAM_USER_UNKNOWN:
    case PAM_ACCT_EXPIRED:
        /* User not allowed in anymore */
        logout();
        break;
    case PAM_NEW_AUTHTOK_REQD:
        /* The user's password has expired. Get a new one */
        do {
            err = pam_chauthtok(pamh, 0);
        } while (err == PAM_AUTHTOK_ERR);
        if (err != PAM_SUCCESS)
            logout();
        break;
    default:
        logout();
    }

    if (pam_setcred(pamh, PAM_REFRESH_CRED) != PAM_SUCCESS){
        logout();
    }

    (void) pam_end(pamh, 0);
    return(0);
    /*NOTREACHED*/

Other Useful PAM Functions

The previous example, Example 3–1, is a simple application that demonstrates only a few of the major PAM functions. This section describes some other PAM functions that can be useful.

The pam_open_session(3PAM) function is called to open a new session after a user has been successfully authenticated.

The pam_getenvlist(3PAM) function is called to establish a new environment. pam_getenvlist() returns a new environment to be merged with the existing environment.
Writing Conversation Functions

A PAM module or application can communicate with a user in a number of ways: command line, dialog box, and so on. As a result, the designer of a PAM consumer that communicates with users needs to write a conversation function. A conversation function passes messages between the user and module independently of the means of communication. A conversation function derives the message type from the msg_style parameter in the conversation function callback pam_message parameter. See pam_start(3PAM).

Developers should make no assumptions about how PAM is to communicate with users. Rather, the application should exchange messages with the user until the operation is complete. Applications should display the message strings for the conversation function without interpretation or modification. An individual message can contain multiple lines, control characters, or extra blank spaces. Note that service modules are responsible for localizing any strings sent to the conversation function.

A sample conversation function, pam_tty_conv(), is provided below. The pam_tty_conv() takes the following arguments:

- **num_msg** – The number of messages that are being passed to the function.
- **mess** – A pointer to the buffer that holds the messages from the user.
- **resp** – A pointer to the buffer that holds the responses to the user.
- **my_data** – Pointer to the application data.

The sample function gets user input from stdin. The routine needs to allocate memory for the response buffer. A maximum, PAM_MAX_NUM_MSG, can be set to limit the number of messages. If the conversation function returns an error, the conversation function is responsible for clearing and freeing any memory that has been allocated for responses. In addition, the conversation function must set the response pointer to NULL. Note that clearing memory should be accomplished using a zero fill approach. The caller of the conversation function is responsible for freeing any responses that have been returned to the caller. To conduct the conversation, the function loops through the messages from the user application. Valid messages are written to stdout, and any errors are written to stderr.

EXAMPLE 3–2  PAM Conversation Function

```c
/*
 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
 * Use is subject to license terms.
 */
#pragma ident "@(#)pam_tty_conv.c 1.4 05/02/12 SMI"
#define __EXTENSIONS__ /* to expose flockfile and friends in stdio.h */
#include <errno.h>
#include <libgen.h>
#include <malloc.h>
#include <signal.h>
#include <stdio.h>
```
EXAMPLE 3-2  PAM Conversation Function  (Continued)

```
#include <stdlib.h>
#include <strings.h>
#include <stropts.h>
#include <unistd.h>
#include <termio.h>
#include <security/pam_appl.h>

static int ctl_c; /* was the conversation interrupted? */

/* ARGUSED 1 */
static void
interrupt(int x)
{
    ctl_c = 1;
}

/* getinput -- read user input from stdin abort on ^C
* Entry  noecho == TRUE, don’t echo input.
* Exit   User’s input.
*        If interrupted, send SIGINT to caller for processing.
* */
static char *
getinput(int noecho)
{
    struct termio tty;
    unsigned short tty_flags;
    char input[PAM_MAX_RESP_SIZE];
    int c;
    int i = 0;
    void (*sig)(int);
    ctl_c = 0;
    sig = signal(SIGINT, interrupt);
    if (noecho) {
        (void) ioctl(fileno(stdin), TCGETA, &tty);
        tty_flags = tty.c_lflag;
        tty.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
        (void) ioctl(fileno(stdin), TCSETAF, &tty);
    }

    /* go to end, but don’t overflow PAM_MAX_RESP_SIZE */
    flockfile(stdin);
    while (ctl_c == 0 &&
        (c = getchar_unlocked()) != '\n' &&
        c != '\r' &&
        c != EOF) {
        if (i < PAM_MAX_RESP_SIZE) {
            input[i++] = (char)c;
        }
    }
    funlockfile(stdin);
    input[i] = '\0';
    if (noecho) {
        tty.c_lflag = tty_flags;
        (void) ioctl(fileno(stdin), TCSETAW, &tty);
        (void) fputc('\n', stdout);
    }
```
EXAMPLE 3–2  PAM Conversation Function  (Continued)

    (void) signal(SIGINT, sig);
    if (ctl_c == 1)
        (void) kill(getpid(), SIGINT);
    return (strdup(input));
}

/* Service modules do not clean up responses if an error is returned. */
/* Free responses here. */
static void
free_resp(int num_msg, struct pam_response *pr)
{
    int i;
    struct pam_response *r = pr;
    if (pr == NULL)
        return;
    for (i = 0; i < num_msg; i++, r++) {
        if (r->resp) {
            /* clear before freeing -- may be a password */
            bzero(r->resp, strlen(r->resp));
            free(r->resp);
            r->resp = NULL;
        }
    }
    free(pr);
}

/* ARGSUSED */
int
pam_tty_conv(int num_msg, struct pam_message **mess,
              struct pam_response **resp, void *my_data)
{
    struct pam_message *m = *mess;
    struct pam_response *r;
    int i;
    if (num_msg <= 0 || num_msg >= PAM_MAX_NUM_MSG) {
        (void) fprintf(stderr, "bad number of messages %d
"
            "<= 0 || >= %d\n", num_msg, PAM_MAX_NUM_MSG);
        *resp = NULL;
        return (PAM_CONV_ERR);
    }
    if ((r = calloc(num_msg, sizeof (struct pam_response))) == NULL)
        return (PAM_BUF_ERR);
    /* Loop through messages */
    for (i = 0; i < num_msg; i++) {
        int echo_off;
/* bad message from service module */
if (m->msg == NULL) {
    (void) fprintf(stderr, "message[%d]: %d/NULL\n", i, m->msg_style);
    goto err;
}

/* fix up final newline: * removed for prompts * added back for messages */
if (m->msg[strlen(m->msg)] == '\n')
    m->msg[strlen(m->msg)] = '\0';

r->resp = NULL;
r->resp_retcode = 0;
echo_off = 0;
switch (m->msg_style) {
    case PAM_PROMPT_ECHO_OFF:
        echo_off = 1;
        /*FALLTHROUGH*/
    case PAM_PROMPT_ECHO_ON:
        (void) fputs(m->msg, stdout);
        r->resp = getinput(echo_off);
        break;
    case PAM_ERROR_MSG:
        (void) fputs(m->msg, stderr);
        (void) fputc('\n', stderr);
        break;
    case PAM_TEXT_INFO:
        (void) fputs(m->msg, stdout);
        (void) fputc('\n', stdout);
        break;
    default:
        (void) fprintf(stderr, "message[%d]: unknown type ",
                       %d/val="%s\n", i, m->msg_style, m->msg);
        /* error, service module won’t clean up */
        goto err;
    }
    if (errno == EINTR)
        goto err;

    /* next message/response */
    m++;
    r++;
}
return (PAM_SUCCESS);
Writing Modules That Provide PAM Services

This section describes how to write PAM service modules.

Requirements for PAM Service Providers

PAM service modules use `pam_get_item(3PAM)` and `pam_set_item(3PAM)` to communicate with applications. To communicate with each other, service modules use `pam_get_data(3PAM)` and `pam_set_data(3PAM)`. If service modules from the same project need to exchange data, then a unique data name for that project should be established. The service modules can then share this data through the `pam_get_data()` and `pam_set_data()` functions.

Service modules must return one of three classes of PAM return code:

- `PAM_SUCCESS` if the module has made a positive decision that is part of the requested policy.
- `PAM_IGNORE` if the module does not make a policy decision.
- `PAM_ERROR` if the module participates in the decision that results in a failure. The `error` can be either a generic error code or a code specific to the service module type. The error cannot be an error code for another service module type. See the specific man page for `pam_sm_module-type` for the error codes.

If a service module performs multiple functions, these functions should be split up into separate modules. This approach gives system administrators finer-grained control for configuring policy.

Man pages should be provided for any new service modules. Man pages should include the following items:

- Arguments that the module accepts.
- All functions that the module implements.
- The effect of flags on the algorithm.
- Any required PAM items.
- Error returns that are specific to this module.

EXAMPLE 3–2  PAM Conversation Function  (Continued)

```c
err:
   free_resp(i, r);
   *resp = NULL;
   return (PAM_CONV_ERR);
}
```
Service modules are required to honor the PAM_SILENT flag for preventing display of messages. The debug argument is recommended for logging debug information to syslog. Use `syslog(3C)` with LOG_AUTH and LOG_DEBUG for debug logging. Other messages should be sent to `syslog()` with LOG_AUTH and the appropriate priority. `openlog(3C)`, `closelog(3C)`, and `setlogmask(3C)` must not be used as these functions interfere with the applications settings.

**Sample PAM Provider Service Module**

A sample PAM service module follows. This example checks to see if the user is a member of a group that is permitted access to this service. The provider then grants access on success or logs an error message on failure. The example goes through the following steps:

1. Parse the options passed to this module from the configuration line in `/etc/pam.conf`

   This module accepts the `nowarn` and `debug` options as well as a specific option `group`. With the `group` option, the module can be configured to allow access for a particular group other than the group `root` that is used by default. See the definition of DEFAULT_GROUP in the source code for the example. For example, to allow telnet(1) access by users that belong to group `staff`, one could use the following line, which is in the telnet stack in `/etc/pam.conf`:
   ```
telnet account required pam_members_only.so.1 group=staff
   ```

2. Get the username, service name and hostname.

   The username is obtained by calling `pam_get_user(3PAM)` which retrieves the current user name from the PAM handle. If the username has not been set, access is denied. The service name and the host name are obtained by calling `pam_get_item(3PAM)`.

3. Validate the information to be worked on.

   If the user name is not set, deny access. If the group to be worked on is not defined, deny access.

4. Verify that the current user is a member of the special group that allows access to this host and grant access.

   In the event that the special group is defined but contains no members at all, PAM_IGNORE is returned to indicate that this module does not participate in any account validation process. The decision is left to other modules on the stack.

5. If the user is not a member of the special group, display a message to inform the user that access is denied.

   Log a message to record this event.

The following example shows the source code for the sample PAM provider.
EXAMPLE 3-3  Sample PAM Service Module

/*
 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
 * Use is subject to license terms.
 */

#include <stdio.h>
#include <stdlib.h>
#include <grp.h>
#include <string.h>
#include <syslog.h>
#include <libintl.h>
#include <security/pam_appl.h>

/* by default, only users who are a member of group "root" are allowed access */
#define DEFAULT_GROUP "root"

static char *NOMSG =
"Sorry, you are not on the access list for this host - access denied."

int
pam_sm_acct_mgmt(pam_handle_t * pamh, int flags, int argc, const char **argv)
{
    char *user = NULL;
    char *host = NULL;
    char *service = NULL;
    const char *allowed_grp = DEFAULT_GROUP;
    char grp_buf[4096];
    struct group grp;
    struct pam_conv *conversation;
    struct pam_message message;
    struct pam_message *pmessage = &message;
    struct pam_response *res = NULL;
    int i;
    int nowarn = 0;
    int debug = 0;

    /* Set flags to display warnings if in debug mode. */
    for (i = 0; i < argc; i++) {
        if (strcasecmp(argv[i], "nowarn") == 0) {
            nowarn = 1;
        } else if (strcasecmp(argv[i], "debug") == 0) {
            debug = 1;
        } else if (strncmp(argv[i], "group=", 6) == 0) {
            allowed_grp = &argv[i][6];
        }
    }
    if (flags & PAM_SILENT) {
        nowarn = 1;
    }

    /* Get user name, service name, and host name. */
    (void) pam_get_user(pamh, &user, NULL);
    (void) pam_get_item(pamh, PAM_SERVICE, (void **) &service);
    (void) pam_get_item(pamh, PAM_RHOST, (void **) &host);

    /* Deny access if user is NULL. */
    if (user == NULL) {

EXAMPLE 3–3 Sample PAM Service Module (Continued)

    syslog(LOG_AUTH|LOG_DEBUG,
        "%s: members only: user not set", service);
    return (PAM_USER_UNKNOWN);
}

if (host == NULL)
    host = "unknown";

/* Deny access if vuser group is required and user is not in vuser
 * group */
if (getgrnam_r(allowed_grp, &grp, grp_buf, sizeof (grp_buf)) == NULL) {
    syslog(LOG_NOTICE|LOG_AUTH,
        "%s: members only: group \"%s\" not defined", 
            service, allowed_grp);
    return (PAM_SYSTEM_ERR);
}

/* Ignore this module if group contains no members. */
if (grp.gr_mem[0] == 0) {
    if (debug)
        syslog(LOG_AUTH|LOG_DEBUG,
            "%s: members only: group %s empty: "
            "all users allowed.", service, grp.gr_name);
    return (PAM_IGNORE);
}

/* Check to see if user is in group. If so, return SUCCESS. */
for (; grp.gr_mem[0]; grp.gr_mem++) {
    if (strcmp(grp.gr_mem[0], user) == 0) {
        if (debug)
            syslog(LOG_AUTH|LOG_DEBUG,
                "%s: user %s is member of group %s. 
                    Access allowed.", service, user, grp.gr_name);
        return (PAM_SUCCESS);
    }
}

/* User is not a member of the group. 
 * Set message style to error and specify denial message. */
message.msg_style = PAM_ERROR_MSG;
message.msg = gettext(NOMSG);
/* Use conversation function to display denial message to user. */
(void) pam_get_item(pamh, PAM_CONV, (void **) &conversation);
if (nowarn == 0 && conversation != NULL) {
    int err;
    err = conversation->conv(1, &pmessage, &res,
                conversation->appdata_ptr);
    if (debug && err != PAM_SUCCESS)
        syslog(LOG_AUTH|LOG_DEBUG,
            "%s: members only: conversation returned "
            "error %d", service, err);
EXAMPLE 3-3 Sample PAM Service Module (Continued)

"error %d (%s).", service, err,
pam_strerror(pamh, err));

/* free response (if any) */
if (res != NULL) {
    if (res->resp)
        free(res->resp);
    free(res);
}
}

/* Report denial to system log and return error to caller. */
syslog(LOG_NOTICE | LOG_AUTH, "%s: members only: *
    "Connection for %s not allowed from %s", service, user, host);

return (PAM_PERM_DENIED);
Introduction to GSS-API

GSS-API enables programmers to write applications generically with respect to security. Developers do not have to tailor the security implementations to any particular platform, security mechanism, type of protection, or transport protocol. With GSS-API, a programmer can avoid the details of protecting network data. A program that uses GSS-API is more portable with regards to network security. This portability is the hallmark of the Generic Security Service API.

GSS-API is a framework that provides security services to callers in a generic fashion. The GSS-API framework is supported by a range of underlying mechanisms and technologies, such as Kerberos v5 or public key technologies, as shown in the following figure.
Broadly speaking, GSS-API does two main things:

1. GSS-API creates a security context in which data can be passed between applications. A context is a state of trust between two applications. Applications that share a context recognize each other and thus can permit data transfers while the context lasts.

2. GSS-API applies one or more types of protection, known as security services, to the data to be transmitted. Security services are explained in “Security Services in GSS-API” on page 57.

In addition, GSS-API performs the following functions:

- Data conversion
- Error checking
- Delegation of user privileges
- Information display
- Identity comparison

GSS-API includes numerous support and convenience functions.

**Application Portability With GSS-API**

GSS-API provides several types of portability for applications:

- **Mechanism independence.** GSS-API provides a generic interface for security. By specifying a default security mechanism, an application does not need to know the mechanism to be applied nor any details about that mechanism.

- **Protocol independence.** GSS-API is independent of any communications protocol or protocol suite. For example, GSS-API can be used with applications that use sockets, RCP, or TCP/IP.
RPCSEC_GSS is an additional layer that smoothly integrates GSS-API with RPC. For more information, see "Remote Procedure Calls With GSS-API" on page 57.

- **Platform independence.** GSS-API is independent of the type of operating system on which an application is running.
- **Quality of Protection independence.** Quality of Protection (QOP) refers to the type of algorithm for encrypting data or generating cryptographic tags. GSS-API allows a programmer to ignore QOP by using a default that is provided by GSS-API. On the other hand, an application can specify the QOP if necessary.

### Security Services in GSS-API

GSS-API provides three types of security services:

- **Authentication** – Authentication is the basic security offered by GSS-API. Authentication is the verification of an identity. If a user is authenticated, the system assumes that person is the one who is entitled to operate under that username.

- **Integrity** – Integrity is the verification of the data’s validity. Even if data comes from a valid user, the data itself could have become corrupted or compromised. Integrity ensures that a message is complete as intended, with nothing added and nothing missing. GSS-API provides for data to be accompanied by a cryptographic tag, known as an Message Integrity Code (MIC). The MIC proves that the data that you receive is the same as the data that the sender transmitted.

- **Confidentiality** – Confidentiality ensures that a third party who intercepted the message would have a difficult time reading the contents. Neither authentication nor integrity modify the data. If the data is somehow intercepted, others can read that data. GSS-API therefore allows data to be encrypted, provided that underlying mechanisms are available that support encryption. This encryption of data is known as confidentiality.

### Available Mechanisms in GSS-API

The current implementation of GSS-API works with the following mechanisms: Kerberos v5, Diffie-Hellman, and SPNEGO. For more information on the Kerberos implementation, see Chapter 21, "Introduction to the Kerberos Service," in *System Administration Guide: Security Services* for more information. Kerberos v5 should be installed and running on any system on which GSS-API-aware programs are running.

### Remote Procedure Calls With GSS-API

Programmers who use the RPC (Remote Procedure Call) protocol for networking applications can use RPCSEC_GSS to provide security. RPCSEC_GSS is a separate layer that sits on top of GSS-API. RPCSEC_GSS provides all the functionality of GSS-API in a way that is tailored to
RPC. In fact, RPCSEC_GSS serves to hide many aspects of GSS-API from the programmer, making RPC security especially accessible and portable. For more information about RPCSEC_GSS, see “Authentication Using RPCSEC_GSS” in ONC+ Developer’s Guide.

The following diagram illustrates how the RPCSEC_GSS layer sits between the application and GSS-API.

**FIGURE 4–2** RPCSEC_GSS and GSS-API

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPCSEC_GSS</td>
</tr>
<tr>
<td>GSS-API</td>
</tr>
<tr>
<td>Kerberos V5</td>
</tr>
<tr>
<td>RSA Public Key</td>
</tr>
<tr>
<td>Other...</td>
</tr>
</tbody>
</table>

**Limitations of GSS-API**

Although GSS-API makes protecting data simple, GSS-API avoids some tasks that would not be consistent with GSS-API’s generic nature. Accordingly, GSS-API does not perform the following activities:

- Provide security credentials for users or applications. Credentials must be provided by the underlying security mechanisms. GSS-API does allow applications to acquire credentials, either automatically or explicitly.
- Transfer data between applications. The application has the responsibility for handling the transfer of all data between peers, whether the data is security-related or plain data.
- Distinguish between different types of transmitted data. For example, GSS-API does not know whether a data packet is plain data or encrypted.
- Indicate status due to asynchronous errors.
- Protect by default information that has been sent between processes of a multiprocess program.
- Allocate string buffers to be passed to GSS-API functions. See “Strings and Similar Data in GSS-API” on page 59.
- Deallocate GSS-API data spaces. This memory must be explicitly deallocated with functions such as gss_release_buffer() and gss_delete_name().
Language Bindings for GSS-API

This document currently covers only the C language bindings, that is, functions and data types, for GSS-API. A Java-bindings version of GSS-API is now available. The Java GSS-API contains the Java bindings for the Generic Security Services Application Program Interface (GSS-API), as defined in RFC 2853.

Where to Get More Information on GSS-API

These two documents provide further information about GSS-API:


Important Elements of GSS-API

This section covers the following important GSS-API concepts: principals, GSS-API data types, status codes, and tokens.

- “GSS-API Data Types” on page 59
- “GSS-API Status Codes” on page 68
- “GSS-API Tokens” on page 69

GSS-API Data Types

The following sections explain the major GSS-API data types. For information on all GSS-API data types, see “GSS-API Data Types and Values” on page 221.

GSS-API Integers

Because the size of an int can vary from platform to platform, GSS-API provides the following integer data type: OM_uint32 which is a 32-bit unsigned integer.

Strings and Similar Data in GSS-API

Because GSS-API handles all data in internal formats, strings must be converted to a GSS-API format before being passed to GSS-API functions. GSS-API handles strings with the gss_buffer_desc structure:
typedef struct gss_buffer_desc_struct {
    size_t length;
    void *value;
} gss_buffer_desc *gss_buffer_t;

gss_buffer_t is a pointer to such a structure. Strings must be put into a gss_buffer_desc structure before being passed to functions that use them. In the following example, a generic GSS-API function applies protection to a message before sending that message.

**EXAMPLE 4-1 Using Strings in GSS-API**

```
char *message_string;
gss_buffer_desc input_msg_buffer;

input_msg_buffer.value = message_string;
ininput_msg_buffer.length = strlen(input_msg_buffer.value) + 1;
gss_generic_function(arg1, &input_msg_buffer, arg2...);
gss_release_buffer(input_msg_buffer);
```

Note that `input_msg_buffer` must be deallocated with `gss_release_buffer()` when you are finished with `input_msg_buffer`.

The `gss_buffer_desc` object is not just for character strings. For example, tokens are manipulated as `gss_buffer_desc` objects. See “GSS-API Tokens” on page 69 for more information.

**Names in GSS-API**

A *name* refers to a principal. In network-security terminology, a *principal* is a user, a program, or a machine. Principals can be either clients or servers. Some examples of principals follows:

- A user, such as `user@machine`, who logs into another machine
- A network service, such as `nfs@machine`
- A machine, such as `myHost@eng.company.com`, that runs an application

In GSS-API, names are stored as a `gss_name_t` object, which is opaque to the application. Names are converted from `gss_buffer_t` objects to the `gss_name_t` form by the `gss_import_name()` function. Every imported name has an associated *name type*, which indicates the format of the name. See “GSS-API OIDs” on page 66 for more about name types. See “Name Types” on page 222 for a list of valid name types.

`gss_import_name()` has the following syntax:

```
OM_uint32 gss_import_name (  
    OM_uint32 *minor-status,  
    const gss_buffer_t input-name-buffer,  
    const gss_OID input-name-type,  
    gss_name_t *output-name)
```
Important Elements of GSS-API

**minor-status**
Status code returned by the underlying mechanism. See “GSS-API Status Codes” on page 68.

**input-name-buffer**
The gss_buffer_desc structure containing the name to be imported. The application must allocate this structure explicitly. See “Strings and Similar Data in GSS-API” on page 59 as well as Example 4–2. This argument must be deallocated with gss_release_buffer() when the application is finished with the space.

**input-name-type**
A gss_OID that specifies the format of input-name-buffer. See “Name Types in GSS-API” on page 67. Also, “Name Types” on page 222 contains a table of valid name types.

**output-name**
The gss_name_t structure to receive the name.

A minor modification of the generic example shown in Example 4–1 illustrates how gss_import_name() can be used. First, the regular string is inserted into a gss_buffer_desc structure. Then gss_import_name() places the string into a gss_name_t structure.

**EXAMPLE 4–2** Using gss_import_name()

```c
char *name_string;
gss_buffer_desc input_name_buffer;
gss_name_t output_name_buffer;

input_name_buffer.value = name_string;
input_name_buffer.length = strlen(input_name_buffer.value) + 1;
gss_import_name(&minor_status, input_name_buffer, GSS_C_NT_HOSTBASED_SERVICE, &output_name);
gss_release_buffer(input_name_buffer);
```

An imported name can be put back into a gss_buffer_t object for display in human-readable form with gss_display_name(). However, gss_display_name() does not guarantee that the resulting string will be the same as the original due to the way the underlying mechanisms store names. GSS-API includes several other functions for manipulating names. See “GSS-API Functions” on page 215.

A gss_name_t structure can contain several versions of a single name. One version is produced for each mechanism that is supported by GSS-API. That is, a gss_name_t structure for user@company might contain one version of that name as rendered by Kerberos v5 and another version that was given by a different mechanism. The function gss_canonicalize_name() takes as input an internal name and a mechanism. gss_canonicalize_name() yields a second internal name that contains a single version of the name that is specific to that mechanism.
Such a mechanism-specific name is called a *mechanism name* (MN). A mechanism name does not refer to the name of a mechanism, but to the name of a principal as produced by a given mechanism. This process is illustrated in the following figure.

**FIGURE 4-3  Internal Names and Mechanism Names**

```
joe@machine  string
```

```
gss_import_name()
```

```
joe@machine  mech1
```

```
joe@machine  mech2
```

```
joe@machine  mech\text{n}
```

```
gss_canonicalize_name(mech2)
```

```
joe@machine  mech2
```

**internal name format**

\((\text{gss\_name\_t})\)

**internal name format,**

\(\text{MN (Mechanism Name)}\)

\((\text{gss\_name\_t})\)
Comparing Names in GSS-API

Consider the case where a server has received a name from a client and needs to look up that name in an access control list. An access control list, or ACL, is a list of principals with particular access permissions. One way to do the lookup would be as follows:

1. Import the client name into GSS-API internal format with `gss_import_name()`, if the name has not already been imported.
   
   In some cases, the server will receive a name in internal format, so this step will not be necessary. For example, a server might look up the client's own name. During context initiation, the client's own name is passed in internal format.

2. Import each name in the ACL with `gss_import_name()`.

3. Compare each imported ACL name with the imported client's name, using `gss_compare_name()`.

This process is shown in the following figure. In this case, Step 1 is assumed to be needed.
FIGURE 4–4  Comparing Names (Slow)

SERVER

(From Client)

Client Name

1. gss_import_name()

Client Name (As internal name)

3. gss_compare_name()

2. gss_import_name()

ACL Name 1 (As internal name)  ACL Name 2 (As internal name)  ACL Name n (As internal name)

1 2 3

Important Elements of GSS-API
The previous approach of comparing names individually is acceptable when there are only a few names. When there are a large number of names, using the gss_canonicalize_name() function is more efficient. This approach uses the following steps.

1. Import the client's name with gss_import_name(), if the name has not already been imported.
   As with the previous method of comparing names, if the name is already in internal format, this step is unnecessary.
2. Use gss_canonicalize_name() to produce a mechanism name version of the client's name.
3. Use gss_export_name() to produce an exported name, which is the client's name as a contiguous string.
4. Compare the exported client's name with each name in the ACL by using memcmp(), which is a fast, low-overhead function.

This process is shown in the following figure. Again, assume that the server needs to import the name that is received from the client.
Because \texttt{gss\_export\_name()} expects a mechanism name (MN), you\textit{must} run \texttt{gss\_canonicalize\_name()} on the client's name first.

See the \texttt{gss\_export\_name(3GSS)}, \texttt{gss\_import\_name(3GSS)}, and \texttt{gss\_canonicalize\_name(3GSS)} for more information.

**GSS-API OIDs**

Object identifiers (OIDs) are used to store the following kinds of data:

- Security mechanisms
- QOPs – Quality of Protection values
Name types

OIDs are stored in GSS-API gss_OID_desc structure. GSS-API provides a pointer to the structure, gss_OID, as shown in the following example.

**EXAMPLE 4-3  OIDs Structure**
```c
typedef struct gss_OID_desc_struct {
    OM_uint32 length;
    void *elements;
} gss_OID_desc, *gss_OID;
```

Further, one or more OIDs might be contained in a gss_OID_set_desc structure.

**EXAMPLE 4-4  OID Set Structure**
```c
typedef struct gss_OID_set_desc_struct {
    size_t count;
    gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

**Caution** – Applications should not attempt to deallocate OIDs with `free()`.

Mechanisms and QOPs in GSS-API

Although GSS-API allows applications to choose underlying security mechanisms, applications should use the default mechanism that has been selected by GSS-API if possible. Similarly, although GSS-API lets an application specify a Quality of Protection level for protecting data, the default QOP should be used if possible. Acceptance of the default mechanism is indicated by passing the value `GSS_C_NULL_OID` to functions that expect a mechanism or QOP as an argument.

**Caution** – Specifying a security mechanism or QOP explicitly defeats the purpose of using GSS-API. Such a specific selection limits the portability of an application. Other implementations of GSS-API might not support that QOP or mechanism in the intended manner. Nonetheless, Appendix C, “Specifying an OID,” briefly discusses how to find out which mechanisms and QOPs are available, and how to choose one.

Name Types in GSS-API

Besides QOPs and security mechanisms, OIDs are also used to indicate name types, which indicate the format for an associated name. For example, the function `gss_import_name()`, which converts the name of a principal from a string to a gss_name_t type, takes as one
argument the format of the string to be converted. If the name type is, for example, GSS_C_NT_HOSTBASED_SERVICE, then the function knows that the name being input is of the form service@host. If the name type is GSS_C_NT_EXPORT_NAME, then the function expects a GSS-API exported name. Applications can find out which name types are available for a given mechanism with the gss_inquire_names_for_mech() function. A list of name types used by GSS-API is provided in “Name Types” on page 222.

GSS-API Status Codes

All GSS-API functions return two types of codes that provide information on the function’s success or failure. Both types of status codes are returned as OM_uint32 values. The two types of return codes are as follows:

- **Major status codes** – Codes that indicate the following errors:
  - Generic GSS-API routine errors, such as giving a routine an invalid mechanism
  - Call errors that are specific to a particular GSS-API language binding, such as a function argument that cannot be read, cannot be written, or is malformed
  - Both types of errors

  Additionally, major status codes can provide supplementary information about a routine’s status. For example, a code might indicate that an operation is not finished, or that a token has been sent out of order. If no errors occur, the routine returns a major status value of GSS_S_COMPLETE.

  Major status codes are returned as follows:

  ```c
  OM_uint32 major_status ; /* status returned by GSS-API */
  major_status = gss_generic_function(arg1, arg2 ...);
  ```

  Major status return codes can be processed like any other OM_uint32. For example, consider the following code.

  ```c
  OM_uint32 maj_stat;
  maj_stat = gss_generic_function(arg1, arg2 ...);
  if (maj_stat == GSS_CREDENTIALS_EXPIRED)
      <do something...>
  ```

  Major status codes can be processed with the macros GSS_ROUTINE_ERROR(), GSS_CALLING_ERROR(), and GSS_SUPPLEMENTARY_INFO(). “GSS-API Status Codes” on page 217 explains how to read major status codes and contains a list of GSS-API status codes.

- **Minor status codes** – Codes that are returned by the underlying mechanism. These codes are not specifically documented in this manual.
Every GSS-API function has as a first argument an *OM_uint32* type for the minor code status. The minor status code is stored in the *OM_uint32* argument when the function returns to the calling function. Consider the following code.

```c
OM_uint32 *minor_status ; /* status returned by mech */
major_status = gss_generic_function(&minor_status, arg1, arg2 ...);
```

The *minor_status* parameter is always set by a GSS-API routine, even if a fatal major status code error is returned. Note that most other output parameters can remain unset. However, output parameters that are expected to return pointers to storage that has been allocated by the routine are set to NULL. NULL indicates that no storage was actually allocated. Any length field associated with such pointers, as in a *gss_buffer_desc* structure, are set to zero. In such cases, applications do not need to release these buffers.

### GSS-API Tokens

The basic unit of “currency” in GSS-API is the *token*. Applications that use GSS-API communicate with each other by using tokens. Tokens are used for exchanging data and for making security arrangements. Tokens are declared as *gss_buffer_t* data types. Tokens are opaque to applications.

Two types of tokens are *context-level tokens* and *per-message tokens*. Context-level tokens are used primarily when a context is established, that is, initiated and accepted. Context-level tokens can also be passed afterward to manage a context.

Per-message tokens are used after a context has been established. Per-message tokens are used to provide protection services on data. For example, consider an application that wants to send a message to another application. That application might use GSS-API to generate a cryptographic identifier to go along with that message. The identifier would be stored in a token.

Per-message tokens can be considered with regard to messages as follows. A *message* is a piece of data that an application sends to a peer. For example, the `ls` command could be a message that is sent to an `ftp` server. A *per-message token* is an object generated by GSS-API for that message. A per-message token could be a cryptographic tag or the encrypted form of the message. Note that this last example is mildly inaccurate. An encrypted message is still a message and not a token. A token is only GSS-API-generated information. However, informally, *message* and *per-message* token are often used interchangeably.

An application is responsible for the following activities:

1. Sending and receiving tokens. The developer usually needs to write generalized read and write functions for performing these actions. The *send_token()* and *recv_token()* functions in "Miscellaneous GSS-API Sample Functions" on page 207.
2. Distinguishing between types of tokens and manipulating the tokens accordingly. Because tokens are opaque to applications, the application does not distinguish between one token and another. Without knowing a token's contents, an application must be able to distinguish the token's type to pass that token to an appropriate GSS-API function. An application can distinguish token types through the following methods:

- By state. Through the control-flow of a program. For example, an application that is waiting to accept a context might assume that any received tokens are related to context establishment. Peers are expected to wait until the context is fully established before sending message tokens, that is, data. After the context is established, the application assumes that new tokens are message tokens. This approach to handling tokens is a fairly common way to handle tokens. The sample programs in this book use this method.

- By flags. For example, if an application has a function for sending tokens to peers, that application can include a flag to indicate the kind of token. Consider the following code:

```c
#include <gssapi.h>

void send_a_token(gss_buffer_t token)
{
    gss_buffer_t token;
    gss_buffer_t token_flag;
    token_flag = MIC_TOKEN;
    send_a_token(&token, token_flag);
}
```

The receiving application would have a receiving function, for example, `get_a_token()`, that would check the `token_flag` argument.

- Through explicit tagging. Applications can use meta-tokens. A meta-token is a user-defined structure that contain tokens that have been received from GSS-API functions. A meta-token includes user-defined fields that signal how the tokens that are provided by GSS-API are to be used.

### Interprocess Tokens in GSS-API

GSS-API permits a security context to be passed from one process to another in a multiprocess application. Typically, a application has accepted a client's context. The application then shares the context among that application's processes. See “Exporting and Importing Contexts in GSS-API” on page 80 for information about multiprocess applications.

The `gss_export_context()` function creates an interprocess token. This token contains information that enables the context to be reconstituted by a second process. The application is responsible for passing the interprocess token from one process to the other. This situation is similar to the application's responsibility for passing tokens to other applications.

The interprocess token might contain keys or other sensitive information. Not all GSS-API implementations cryptographically protect interprocess tokens. Therefore, the application must protect interprocess tokens before an exchange takes place. This protection might involve encrypting the tokens with `gss_wrap()`, if encryption is available.
Developing Applications That Use GSS-API

This section shows how to implement secure data exchange using GSS-API. The section focuses on those functions that are most central to using GSS-API. For more information, see Appendix B, “GSS-API Reference,” which contains a list of all GSS-API functions, status codes, and data types. To find out more about any GSS-API function, check the individual man page.

The examples in this manual follow a simple model. A client application sends data directly to a remote server. No mediation by transport protocol layers such as RPC occurs.

Generalized GSS-API Usage

The general steps for using GSS-API are as follows:

1. Each application, both sender and recipient, acquires credentials explicitly, unless credentials have been acquired automatically.
2. The sender initiates a security context. The recipient accepts the context.
3. The sender applies security protection to the data to be transmitted. The sender either encrypts the message or stamps the data with an identification tag. The sender then transmits the protected message.

   Note – The sender can choose not to apply either security protection, in which case the message has only the default GSS-API security service, that is, authentication.

4. The recipient decrypts the message if needed and verifies the message if appropriate.
5. (Optional) The recipient returns an identification tag to the sender for confirmation.
6. Both applications destroy the shared security context. If necessary, the allocations can also deallocate any remaining GSS-API data.

Caution – The calling application is responsible for freeing all data space that has been allocated.

Applications that use GSS-API need to include the file gssapi.h.

Note – Do not assume that interprocess tokens are transferable across different GSS-API implementations.
Working With Credentials in GSS-API

A credential is a data structure that provides proof of an application’s claim to a principal name. An application uses a credential to establish that application’s global identity. Additionally, a credential may be used to confirm an entity’s privileges.

GSS-API does not provide credentials. Credentials are created by the security mechanisms that underly GSS-API, before GSS-API functions are called. In many cases, a user receives credentials at login.

A given GSS-API credential is valid for a single principal. A single credential can contain multiple elements for that principal, each created by a different mechanism. A credential that is acquired on a machine with multiple security mechanisms is valid if that credential is transferred to a machine with a subset of those mechanisms. GSS-API accesses credentials through the gss_cred_id_t structure. This structure is called a credential handle. Credentials are opaque to applications. Thus, the application does not need to know the specifics of a given credential.

Credentials come in three forms:

- GSS_C_INITIATE – Identifies applications that only initiate security contexts
- GSS_C_ACCEPT – Identifies applications that only accept security contexts
- GSS_C_BOTH – Identifies applications that can initiate or accept security contexts

Acquiring Credentials in GSS-API

Before a security context can be established, both the server and the client must acquire their respective credentials. A credential can be reused until that credential expires, after which the application must reacquire the credential. Credentials that are used by the client and credentials that are used by the server can have different lifetimes.

GSS-API-based applications can acquire credentials in two ways:

- By using the gss_acquire_cred() or gss_add_cred() function
- By specifying the value GSS_C_NO_CREDENTIAL, which indicates a default credential, when the context is established

In most cases, gss_acquire_cred() is called only by a context acceptor, that is, a server. A context initiator, that is, a client, typically receives credentials at login. A client, therefore, can usually specify the default credential. The server can also bypass gss_acquire_cred() and use that server’s default credential instead.

A client’s credential proves that client’s identity to other processes. A server acquires a credential to enable that server to accept a security context. So when a client makes an ftp request to a server, that client might already have a credential from login. GSS-API
automatically retrieves the credential when the client attempts to initiate a context. The server program, however, explicitly acquires credentials for the requested service (ftp).

If \texttt{gss_acquire_cred()} completes successfully, then \texttt{GSS_S_COMPLETE} is returned. If a valid credential cannot be returned, then \texttt{GSS_S_NO_CRED} is returned. See the \texttt{gss_acquire_cred(3GSS)} man page for other error codes. For an example, see “Acquiring Credentials” on page 110 in Chapter 6, “GSS-API Server Example.”

\texttt{gss_add_cred()} is similar to \texttt{gss_acquire_cred()}. However, \texttt{gss_add_cred()} enables an application to use an existing credential to create a new handle or to add a new credential element. If \texttt{GSS_C_NO_CREDENTIAL} is specified as the existing credential, then \texttt{gss_add_cred()} creates a new credential according to the default behavior. See the \texttt{gss_add_cred(3GSS)} man page for more information.

### Working With Contexts in GSS-API

The two most significant tasks for GSS-API in providing security are to create security contexts and to protect data. After an application acquires the necessary credentials, a security context must be established. To establish a context, one application, typically a client, initiates the context, and another application, usually a server, accepts the context. Multiple contexts between peers are allowed.

The communicating applications establish a joint security context by exchanging authentication tokens. The security context is a pair of GSS-API data structures that contain information to be shared between the two applications. This information describes the state of each application in terms of security. A security context is required for protection of data.

### Initiating a Context in GSS-API

The \texttt{gss_init_sec_context()} function is used to start a security context between an application and a remote peer. If successful, this function returns a context handle for the context to be established and a context-level token to send to the acceptor. Before calling \texttt{gss_init_sec_context()}, the client should perform the following tasks:

1. Acquire credentials, if necessary, with \texttt{gss_acquire_cred()}. Typically, the client receives credentials at login. \texttt{gss_acquire_cred()} can only retrieve initial credentials from the running operating system.

2. Import the name of the server into GSS-API internal format with \texttt{gss_import_name()}. See “Names in GSS-API” on page 60 for more information about names and \texttt{gss_import_name()}.

When calling \texttt{gss_init_sec_context()}, a client typically passes the following argument values:
- GSS_C_NO_CREDENTIAL for the `cred_handle` argument, to indicate the default credential
- GSS_C_NULL_OID for the `mech_type` argument, to indicate the default mechanism
- GSS_C_NO_CONTEXT for the `context_handle` argument, to indicate an initial null context. Because `gss_init_sec_context()` is usually called in a loop, subsequent calls should pass the context handle that was returned by previous calls
- GSS_C_NO_BUFFER for the `input_token` argument, to indicate an initially empty token. Alternatively, the application can pass a pointer to a `gss_buffer_desc` object whose length field has been set to zero
- The name of the server, imported into internal GSS-API format with `gss_import_name()`.

Applications are not bound to use these default values. Additionally, a client can specify requirements for other security parameters with the `req_flags` argument. The full set of `gss_init_sec_context()` arguments is described below.

The context acceptor might require several handshakes to establish a context. That is, an acceptor can require the initiator to send more than one piece of context information before the context is fully established. Therefore, for portability, context initiation should always be done as part of a loop that checks whether the context has been fully established.

If the context is not complete, `gss_init_sec_context()` returns a major status code of GSS_C_CONTINUE_NEEDED. Therefore, a loop should use the return value from `gss_init_sec_context()` to test whether to continue the initiation loop.

The client passes context information to the server in the form of the `output_token`, which is returned by `gss_init_sec_context()`. The client receives information back from the server as an `input_token`. The input token can then be passed as an argument in subsequent calls of `gss_init_sec_context()`. If the received input token has a length of zero, however, then no more output tokens are required by the server.

Therefore, besides checking for the return status of `gss_init_sec_context()`, the loop should check the input token's length. If the length has a nonzero value, another token needs to be sent to the server. Before the loop begins, the input token's length should be initialized to zero. Either set the input token to GSS_C_NO_BUFFER or set the structure's length field to a value of zero.

The following pseudocode demonstrates an example of context establishment from the client side.

```
context = GSS_C_NO_CONTEXT
input token = GSS_C_NO_BUFFER

do

call gss_init_sec_context(credential, context, name, input token,
```
output token, other args...)

if (there's an output token to send to the acceptor)
    send the output token to the acceptor
    release the output token

if (the context is not complete)
    receive an input token from the acceptor

if (there's a GSS-API error)
    delete the context

until the context is complete

A real loop would be more complete with more extensive error-checking. See “Establishing a Security Context With the Server” on page 97 for a real example of such a context-initiation loop. Additionally, the gss_init_sec_context(3GSS) man page provides a less generic example.

In general, the parameter values returned when a context is not fully established are those values that would be returned when the context is complete. See the gss_init_sec_context(3GSS) man page for more information.

If gss_init_sec_context() completes successfully, GSS_S_COMPLETE is returned. If a context-establishment token is required from the peer application, GSS_S_CONTINUE_NEEDED is returned. If errors occur, error codes are returned as shown in the gss_init_sec_context(3GSS) man page.

If context initiation fails, the client should disconnect from the server.

**Accepting a Context in GSS-API**

The other half of context establishment is context acceptance, which is done through the gss_accept_sec_context() function. In a typical scenario, a server accepts a context that has been initiated by a client with gss_init_sec_context().

The main input to gss_accept_sec_context() is an input token from the initiator. The initiator returns a context handle as well as an output token to be returned to the initiator. Before gss_accept_sec_context() can be called, however, the server should acquire credentials for the service that was requested by the client. The server acquires these credentials with the gss_acquire_cred() function. Alternatively, the server can bypass explicit acquisition of credentials by specifying the default credential, that is, GSS_C_NO_CREDENTIAL, when the server calls gss_accept_sec_context().
When calling `gss_accept_sec_context()`, the server can set the following arguments as shown:

- **`cred_handle`** – The credential handle returned by `gss_acquire_cred()`. Alternatively, `GSS_C_NO_CREDENTIAL` can be used to indicate the default credential.
- **`context_handle`** – `GSS_C_NO_CONTEXT` indicates an initial null context. Because `gss_init_sec_context()` is usually called in a loop, subsequent calls should pass the context handle that was returned by previous calls.
- **`input_token`** – The context token received from the client.

The full set of `gss_accept_sec_context()` arguments is described in the following paragraphs.

Security context establishment might require several handshakes. The initiator and acceptor often need to send more than one piece of context information before the context is fully established. Therefore, for portability, context acceptance should always be done as part of a loop that checks whether the context has been fully established. If the context is not yet established, `gss_accept_sec_context()` returns a major status code of `GSS_C_CONTINUE_NEEDED`. Therefore, a loop should use the value that was returned by `gss_accept_sec_context()` to test whether to continue the acceptance loop.

The context acceptor returns context information to the initiator in the form of the output token that was returned by `gss_accept_sec_context()`. Subsequently, the acceptor can receive additional information from the initiator as an input token. The input token is then passed as an argument to subsequent `gss_accept_sec_context()` calls. When `gss_accept_sec_context()` has no more tokens to send to the initiator, an output token with a length of zero is returned. Besides checking for the return status `gss_accept_sec_context()`, the loop should check the output token’s length to see whether another token must be sent. Before the loop begins, the output token’s length should be initialized to zero. Either set the output token to `GSS_C_NO_BUFFER`, or set the structure’s length field to a value of zero.

The following pseudocode demonstrates an example of context establishment from the server side.

```plaintext
context = GSS_C_NO_CONTEXT
output_token = GSS_C_NO_BUFFER

while true:
    receive an input token from the initiator
    call gss_accept_sec_context(context, cred_handle, input_token, output_token, other args...)
    if (there's an output token to send to the initiator)
        send the output token to the initiator
```
release the output token

if (there's a GSS-API error)
    delete the context

until the context is complete

A real loop would be more complete with more extensive error-checking. See “Establishing a Security Context With the Server” on page 97 for a real example of such a context-acceptance loop. Additionally, the gss_accept_sec_context(3GSS) man page provides an example.

Again, GSS-API does not send or receive tokens. Tokens must be handled by the application. Examples of token-transferring functions are found in “Miscellaneous GSS-API Sample Functions” on page 207.

gss_accept_sec_context() returns GSS_S_COMPLETE if it completes successfully. If the context is not complete, the function returns GSS_S_CONTINUE_NEEDED. If errors occur, the function returns error codes. For more information, see the gss_accept_sec_context(3GSS) man page.

Using Other Context Services in GSS-API

The gss_init_sec_context() function enables an application to request additional data protection services beyond basic context establishment. These services are requested through the req_flags argument to gss_init_sec_context().

Not all mechanisms offer all these services. The ret_flags argument for gss_init_sec_context() indicates which services are available in a given context. Similarly, the context acceptor examines the ret_flags value that is returned by gss_init_sec_context() to determine the available services. The additional services are explained in the following sections.

Delegating a Credential in GSS-API

If permitted, a context initiator can request that the context acceptor act as a proxy. In such a case, the acceptor can initiate further contexts on behalf of the initiator.

Suppose someone on Machine A wants to rlogin to Machine B, and then rlogin from Machine B to Machine C. Depending on the mechanism, the delegated credential identifies B either as A or B as proxy for A.

If delegation is permitted, ret_flags can be set to GSS_C_DELEG_FLAG. The acceptor receives a delegated credential as the delegated_cred_handle argument of gss_accept_sec_context(). Delegating a credential is not the same as exporting a context. See “Exporting and Importing
Contexts in GSS-API” on page 80. One difference is that an application can delegate that application’s credentials multiple times simultaneously, while a context can only be held by one process at a time.

**Performing Mutual Authentication Between Peers in GSS-API**

A user who transfers files to an ftp site typically does not need proof of the site’s identity. On the other hand, a user who is required to provide a credit card number to an application would want definite proof of the receiver’s identity. In such a case, *mutual authentication* is required. Both the context initiator and the acceptor must prove their identities.

A context initiator can request mutual authentication by setting the `gss_init_sec_context()` `req_flags` argument to the value `GSS_C_MUTUAL_FLAG`. If mutual authentication has been authorized, the function indicates authorization by setting the `ret_flags` argument to this value. If mutual authentication is requested but not available, the initiating application is responsible for responding accordingly. GSS-API does not automatically terminate a context when mutual authentication is requested but unavailable. Also, some mechanisms always perform mutual authentication even without a specific request.

**Performing Anonymous Authentication in GSS-API**

In normal use of GSS-API, the initiator’s identity is made available to the acceptor as a part of context establishment. However, a context initiator can request that its identity not be revealed to the context acceptor.

For example, consider an application that provides unrestricted access to a medical database. A client of such a service might want to authenticate the service. This approach would establish trust in any information that is retrieved from the database. The client might not want to expose its identity due to privacy concerns, for example.

To request anonymity, set the `req_flags` argument of `gss_init_sec_context()` to `GSS_C_ANON_FLAG`. To verify whether anonymity is available, check the `ret_flags` argument to `gss_init_sec_context()` or `gss_accept_sec_context()` to see whether `GSS_C_ANON_FLAG` is returned.

When anonymity is in effect, calling `gss_display_name()` on a client name that was returned by `gss_accept_sec_context()` or `gss_inquire_context()` produces a generic anonymous name.

**Note** – An application has the responsibility to take appropriate action if anonymity is requested but not permitted. GSS-API does not terminate a context in such a case.
Using Channel Bindings in GSS-API

For many applications, basic context establishment is sufficient to assure proper authentication of a context initiator. In cases where additional security is desired, GSS-API offers the use of channel bindings. Channel bindings are tags that identify the particular data channel that is used. Specifically, channel bindings identify the origin and endpoint, that is, the initiator and acceptor of the context. Because the tags are specific to the originator and recipient applications, such tags offer more proof of a valid identity.

Channel bindings are pointed to by the gss_channel_bindings_t data type, which is a pointer to a gss_channel_bindings_struct structure as shown below.

```c
typedef struct gss_channel_bindings_struct {
  OM_uint32 initiator_addrtype;
  gss_buffer_desc initiator_address;
  OM_uint32 acceptor_addrtype;
  gss_buffer_desc acceptor_address;
  gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

The first two fields are the address of the initiator and an address type that identifies the format in which the initiator’s address is being sent. For example, initiator_addrtype might be sent to GSS_C_AF_INET to indicate that initiator_address is in the form of an Internet address, that is, an IP address. Similarly, the third and fourth fields indicate the address and address type of the acceptor. The final field, application_data, can be used by the application as needed. Set application_data to GSS_C_NO_BUFFER if application_data is not going to be used. If an application does not specify an address, that application should set the address type field to GSS_C_AF_NULLADDR. The “Address Types for Channel Bindings” on page 223 section has a list of valid address type values.

The address types indicate address families rather than specific addressing formats. For address families that contain several alternative address forms, the initiator_address and acceptor_address fields must contain sufficient information to determine which form is used. When not otherwise specified, addresses should be specified in network byte-order, that is, native byte-ordering for the address family.

To establish a context that uses channel bindings, the input_chann_bindings argument for gss_init_sec_context() should point to an allocated channel bindings structure. The structure's fields are concatenated into an octet string, and a MIC is derived. This MIC is then bound to the output token. The application then sends the token to the context acceptor. After receiving the token, the acceptor calls gss_accept_sec_context(). See “Accepting a Context in GSS-API” on page 75 for more information. gss_accept_sec_context() calculates a MIC for the received channel bindings. gss_accept_sec_context() then returns GSS_C_BAD_BINDINGS if the MIC does not match.
Because `gss_accept_sec_context()` returns the transmitted channel bindings, an acceptor can use these values to perform security checking. For example, the acceptor could check the value of `application_data` against code words that are kept in a secure database.

**Note** – An underlying mechanism might not provide confidentiality for channel binding information. Therefore, an application should not include sensitive information as part of channel bindings unless confidentiality is ensured. To test for confidentiality, an application can check the `ret_flags` argument of `gss_init_sec_context()` or `gss_accept_sec_context()`. The values `GSS_C_CONF_FLAG` and `GSS_C_PROT_READY_FLAG` indicate confidentiality. See "Initiating a Context in GSS-API" on page 73 or "Accepting a Context in GSS-API" on page 75 for information on `ret_flags`.

Individual mechanisms can impose additional constraints on the addresses and address types that appear in channel bindings. For example, a mechanism might verify that the `initiator_address` field of the channel bindings to be returned to `gss_init_sec_context()`.

Portable applications should therefore provide the correct information for the address fields. If the correct information cannot be determined, then `GSS_C_AF_NULLADDR` should be specified as the address types.

**Exporting and Importing Contexts in GSS-API**

GSS-API provides the means for exporting and importing contexts. This ability enables a multiprocess application, usually the context acceptor, to transfer a context from one process to another. For example, an acceptor might have one process that listens for context initiators and another that uses the data that is sent in the context. The "Using the `test_import_export_context()` Function" on page 120 section shows how a context can be saved and restored with these functions.

The function `gss_export_sec_context()` creates an interprocess token that contains information about the exported context. See "Interprocess Tokens in GSS-API" on page 70 for more information. The buffer to receive the token should be set to `GSS_C_NO_BUFFER` before `gss_export_sec_context()` is called.

The application then passes the token on to the other process. The new process accepts the token and passes that token to `gss_import_sec_context()`. The same functions that are used to pass tokens between applications can often be used to pass tokens between processes as well.

Only one instantiation of a security process can exist at a time. `gss_export_sec_context()` deactivates the exported context and sets the context handle to `GSS_C_NO_CONTEXT`. `gss_export_sec_context()` also deallocates any process-wide resources that are associated with that context. If the context exportation cannot be completed, `gss_export_sec_context()` leaves the existing security context unchanged and does not return an interprocess token.
Not all mechanisms permit contexts to be exported. An application can determine whether a context can be exported by checking the `ret_flags` argument to `gss_accept_sec_context()` or `gss_init_sec_context()`. If this flag is set to `GSS_C_TRANS_FLAG`, then the context can be exported. (See “Accepting a Context in GSS-API” on page 75 and “Initiating a Context in GSS-API” on page 73.)

Figure 4–6 shows how a multiprocess acceptor might use context exporting to multitask. In this case, Process 1 receives and processes tokens. This step separates the context-level tokens from the data tokens and passes the tokens on to Process 2. Process 2 deals with data in an application-specific way. In this illustration, the clients have already obtained export tokens from `gss_init_sec_context()`. The clients pass the tokens to a user-defined function, `send_a_token()`, which indicates whether the token to be transmitted is a context-level token or a message token. `send_a_token()` transmits the tokens to the server. Although not shown here, `send_a_token()` would presumably be used to pass tokens between threads as well.
FIGURE 4–6 Exporting Contexts: Multithreaded Acceptor Example

- **Multiprocess Server**
  - Process 1 - accepts tokens
  - Process 2 - processes data

- **Send a Token**
  - `gss_accept_sec_context()`
  - `gss_export_sec_context()`
  - `gss_import_sec_context()`

- **Send a Token**
  - `send_a_token()` takes token, flag (data or context) as arguments

- **Client 1**
  - Send a token
  - `gss_accept_sec_context()`
  - `gss_import_sec_context()`

- **Client 2**
  - Send a token
  - `gss_accept_sec_context()`
  - `gss_import_sec_context()`

- **Symbols**
  - = context-level token
  - = data token

- **Legend**
  - `send_a_token()` takes token, flag (data or context) as arguments

- **Events**
  - MSG

- **Notes**
  - If context token
  - If data token
Obtaining Context Information in GSS-API

GSS-API provides a function, `gss_inquire_context(3GSS)`, that obtains information about a given security context. Note that the context does not need to be complete. Given a context handle, `gss_inquire_context()` provides the following information about context:

- Name of the context initiator
- Name of the context acceptor
- Number of seconds for which the context is valid
- Security mechanism to be used with the context
- Several context-parameter flags. These flags are the same as the `ret_flags` argument of the `gss_accept_sec_context(3GSS)` function. The flags cover delegation, mutual authentication, and so on. See "Accepting a Context in GSS-API" on page 75.
- A flag that indicates whether the inquiring application is the context initiator
- A flag that indicates whether the context is fully established

Sending Protected Data in GSS-API

After a context has been established between two peers, a message can be protected before that message is sent.

Establishing a context only uses the most basic GSS-API protection: authentication. Depending on the underlying security mechanisms, GSS-API provides two other levels of protection:

- **Integrity** – A message integrity code (MIC) for the message is generated by the `gss_get_mic()` function. The recipient checks the MIC to ensure that the received message is the same as the message that was sent.
- **Confidentiality** – In addition to using a MIC, the message is encrypted. The GSS-API function `gss_wrap()` performs the encryption.

The difference between `gss_get_mic()` and `gss_wrap()` is illustrated in the following diagram. With `gss_get_mic()`, the receiver gets a tag that indicates the message is intact. With `gss_wrap()`, the receiver gets an encrypted message and a tag.
The function to be used depends on the situation. Because `gss_wrap()` includes the integrity service, many programs use `gss_wrap()`. A program can test for the availability of the confidentiality service. The program can then call `gss_wrap()` with or without confidentiality depending on the availability. An example is "Wrapping and Sending a Message" on page 103. However, because messages that use `gss_get_mic()` do not need to be unwrapped, fewer CPU cycles are used than with `gss_wrap()`. Thus a program that does not need confidentiality might protect messages with `gss_get_mic()`.

**Tagging Messages With `gss_get_mic()`**

Programs can use `gss_get_mic()` to add a cryptographic MIC to a message. The recipient can check the MIC for a message by calling `gss_verify_mic()`.

In contrast to `gss_wrap()`, `gss_get_mic()` produces separate output for the message and the MIC. This separation means that a sender application must arrange to send both the message and the accompanying MIC. More significantly, the recipient must be able to distinguish between the message and the MIC. The following approaches ensure the proper processing of message and MIC:

- Through program control, that is, state. A recipient application might know to call the receiving function twice, once to get a message and a second time to get the message's MIC.
- Through flags. The sender and receiver can flag the kind of token that is included.
- Through user-defined token structures that include both the message and the MIC.
GSS_S_COMPLETE is returned if gss_get_mic() completes successfully. If the specified QOP is not valid, GSS_S_BAD_QOP is returned. For more information, see gss_get_mic(3GSS).

Wrapping Messages With gss_wrap()

Messages can be wrapped by the gss_wrap() function. Like gss_get_mic(), gss_wrap() provides a MIC. gss_wrap() also encrypts a given message if confidentiality is requested and permitted by the underlying mechanism. The message receiver unwraps the message with gss_unwrap().

Unlike gss_get_mic(), gss_wrap() wraps the message and the MIC together in the outgoing message. The function that transmits the bundle need be called only once. On the other end, gss_unwrap() extracts the message. The MIC is not visible to the application.

gss_wrap() returns GSS_S_COMPLETE if the message was successfully wrapped. If the requested QOP is not valid, GSS_S_BAD_QOP is returned. For an example of gss_wrap(), see “Wrapping and Sending a Message” on page 103.

Handling Wrap Size Issues in GSS-API

Wrapping a message with gss_wrap() increases the amount of data to be sent. Because the protected message packet needs to fit through a given transportation protocol, GSS-API provides the function gss_wrap_size_limit(). gss_wrap_size_limit() calculates the maximum size of a message that can be wrapped without becoming too large for the protocol. The application can break up messages that exceed this size before calling gss_wrap(). Always check the wrap-size limit before actually wrapping the message.

The amount of the size increase depends on two factors:

- Which QOP algorithm is used for making the transformation
- Whether confidentiality is invoked

The default QOP can vary from one implementation of GSS-API to another. Thus, a wrapped message can vary in size even if the QOP default is specified. This possibility is illustrated in the following figure.
Regardless of whether confidentiality is applied, `gss_wrap()` still increases the size of a message. `gss_wrap()` embeds a MIC into the transmitted message. However, encrypting the message can further increase the size. The following figure shows this process.
GSS_S_COMPLETE is returned if `gss_wrap_size_limit()` completes successfully. If the specified QOP is not valid, GSS_S_BAD_QOP is returned. “Wrapping and Sending a Message” on page 103 includes an example of how `gss_wrap_size_limit()` can be used to return the maximum original message size.

Successful completion of this call does not necessarily guarantee that `gss_wrap()` can protect a message of length `max-input-size` bytes. This ability depends on the availability of system resources at the time that `gss_wrap()` is called. For more information, see the `gss_wrap_size_limit(3GSS)` man page.

**Detecting Sequence Problems in GSS-API**

As a context initiator transmits sequential data packets to the acceptor, some mechanisms allow the context acceptor to check for proper sequencing. These checks include whether the packets arrive in the right order, and with no unwanted duplication of packets. See following figure. An acceptor checks for these two conditions during the verification of a packet and the unwrapping of a packet. See “Unwrapping the Message” on page 119 for more information.
With `gss_init_sec_context()`, an initiator can check the sequence by applying logical OR to the `req_flags` argument with either `GSS_C_REPLAY_FLAG` or `GSS_C_SEQUENCE_FLAG`. 
Confirming Message Transmission in GSS-API

After the recipient has unwrapped or verified the transmitted message, a confirmation can be returned to the sender. This means sending back a MIC for that message. Consider the case of a message that was not wrapped by the sender but only tagged with a MIC with `gss_get_mic()`. The process, illustrated in Figure 4–9, is as follows:

1. The initiator tags the message with `gss_get_mic()`.
2. The initiator sends the message and MIC to the acceptor.
3. The acceptor verifies the message with `gss_verify_mic()`.
4. The acceptor sends the MIC back to the initiator.
5. The initiator verifies the received MIC against the original message with `gss_verify_mic()`.

**FIGURE 4–9 Confirming MIC Data**

MIC = Message Integrity Code

---

**Diagram Description**

- **CLIENT**
  - Message (MSG)
  - MIC
  - `gss_get_mic()`
  - `gss_verify_mic()`

- **SERVER**
  - Message (MSG)
  - MIC
  - `gss_verify_mic()`
In the case of wrapped data, the `gss_unwrap()` function never produces a separate MIC, so the recipient must generate it from the received and unwrapped message. The process, illustrated in Figure 4–10, is as follows:

1. The initiator wraps the message with `gss_wrap()`.
2. The initiator sends the wrapped message.
3. The acceptor unwraps the message with `gss_unwrap()`.
4. The acceptor calls `gss_get_mic()` to produce a MIC for the unwrapped message.
5. The acceptor sends the derived MIC to the initiator.
6. The initiator compares the received MIC against the original message with `gss_verify_mic()`.

Applications should deallocate any data space that has been allocated for GSS-API data. The relevant functions are `gss_release_buffer(3GSS)`, `gss_release_cred(3GSS)`, `gss_release_name(3GSS)`, and `gss_release_oid_set(3GSS)`. 
Finally, all messages have been sent and received, and the initiator and acceptor applications have finished. At this point, both applications should call `gss_delete_sec_context()` to destroy the shared context. `gss_delete_sec_context()` deletes local data structures that are associated with the context.

For good measure, applications should be sure to deallocate any data space that has been allocated for GSS-API data. The functions that do this are `gss_release_buffer()`, `gss_release_cred()`, `gss_release_name()`, and `gss_release_oid_set()`.

**Cleaning Up a GSS-API Session**
This chapter presents a walk-through of a typical GSS-API client application. The following topics are covered:

- “GSSAPI Client Example Overview” on page 93
- “GSSAPI Client Example: main() Function” on page 95
- “Opening a Connection With the Server” on page 96
- “Establishing a Security Context With the Server” on page 97
- “Miscellaneous GSSAPI Context Operations on the Client Side” on page 102
- “Wrapping and Sending a Message” on page 103
- “Reading and Verifying a Signature Block From a GSS-API Client” on page 105
- “Deleting the Security Context” on page 106

GSSAPI Client Example Overview

The sample client-side program gss-client creates a security context with a server, establishes security parameters, and sends the message string to the server. The program uses a simple TCP-based sockets connection to make the connection.

The following sections provide a step-by-step description of how gss-client works. Because gss-client is a sample program that has been designed to show off GSSAPI functionality, only relevant parts of the program are discussed in detail. The complete source code for the two applications appears in the appendix and can be downloaded from


GSSAPI Client Example Structure

The gss-client application performs the following steps:

1. Parses the command line.
2. Creates an object ID (OID) for a mechanism, if a mechanism is specified. Otherwise, the default mechanism is used, which is most commonly the case.
3. Creates a connection to the server.
4. Establishes a security context.
5. Wraps and sends the message.
6. Verifies that the message has been “signed” correctly by the server.
7. Deletes the security context.

Running the GSSAPI Client Example

The gss-client example takes this form on the command line:

```
gss-client [-port port] [-d] [-mech mech] host service-name [-f] msg
```

- `port` – The port number for making the connection to the remote machine that is specified by `host`.
- `-d` flag – Causes security credentials to be delegated to the server. Specifically, the `deleg-flag` variable is set to the GSS-API value `GSS_C_DELEG_FLAG`. Otherwise, `deleg-flag` is set to zero.
- `mech` – The name of the security mechanism, such as Kerberos v5 to be used. If no mechanism is specified, the GSS-API uses a default mechanism.
- `host` – The name of the server.
- `service-name` – The name of the network service requested by the client. Some typical examples are the `telnet`, `ftp`, and `login` services.
- `msg` – The string to send to the server as protected data. If the `-f` option is specified, then `msg` is the name of a file from which to read the string.

A typical command line for client application program might look like the following example:

```
gss-client -port 8080 -d -mech kerberos_v5 erebos.eng nfs "ls"
```

The following example does not specify a mechanism, port, or delegation:

```
gss-client erebos.eng nfs "ls"
```
GSSAPI Client Example: main() Function

As with all C programs, the outer shell of the program is contained in the entry-point function, `main()`. main() performs four functions:

- Parses command-line arguments and assigns the arguments to variables.
- Calls `parse_oid()` to create a GSS-API OID, object identifier, if a mechanism other than the default is to be used. The object identifier comes from the name of the security mechanism, provided that a mechanism name has been supplied.
- Calls `call_server()`, which does the actual work of creating a context and sending data.
- Releases the storage space for the OID if necessary, after the data is sent.

The source code for the `main()` routine is shown in the following example.

```c
EXAMPLE 5-1 gss-client Example: main()
int main(argc, argv)
   int argc;
   char **argv;
{
   char *msg;
   char service_name[128];
   char hostname[128];
   char *mechanism = 0;
   u_short port = 4444;
   int use_file = 0;
   OM_uint32 deleg_flag = 0, min_stat;
   display_file = stdout;

   /* Parse command-line arguments. */
   argc--; argv++;
   while (argc) {
      if (strcmp(*argv, "-port") == 0) {
         argc--; argv++;
         port = atoi(*argv);
      } else if (strcmp(*argv, "-mech") == 0) {
         argc--; argv++;
         mechanism = *argv;
      } else if (strcmp(*argv, "-d") == 0) {
         deleg_flag = GSS_C_DELEG_FLAG;
      } else if (strcmp(*argv, "-f") == 0) {
         use_file = 1;
      } else
         break;
      argc--; argv++;
   }
   if (argc != 3)
      usage();
   if (argc > 1) {
      /* Process non-option arguments. */
```
EXAMPLE 5–1  gss-client Example: main()  (Continued)

    strcpy(hostname, argv[0]);
} else if (gethostname(hostname, sizeof(hostname)) == -1) {
    perror("gethostname");
    exit(1);
}

if (argc > 2) {
    strcpy(service_name, argv[1]);
    strcat(service_name, "@");
    strcat(service_name, hostname);
}

msg = argv[2];

/* Create GSSAPI object ID. */
if (mechanism)
    parse_oid(mechanism, &g_mechOid);

/* Call server to create context and send data. */
if (call_server(hostname, port, g_mechOid, service_name,
                deleg_flag, msg, use_file) < 0)
    exit(1);

/* Release storage space for OID, if still allocated */
if (g_mechOid != GSS_C_NULL_OID)
    (void) gss_release_oid(&min_stat, &gmechOid);
return 0;
}

Opening a Connection With the Server

The call_server() function uses the following code to make the connection with the server:

    if ((s = connect_to_server(host, port)) < 0)
        return -1;

s is a file descriptor, the int that is initially returned by a call to socket().
connect_to_server() is a simple function outside GSS-API that uses sockets to create a
connection. The source code for connect_to_server() is shown in the following example.

EXAMPLES 5–2  connect_to_server() Function

int connect_to_server(host, port)
    char *host;
    u_short port;
{
    struct sockaddr_in saddr;

EXAMPLE 5–2  connect_to_server() Function  (Continued)

```c
struct hostent *hp;
int s;

if ((hp = gethostbyname(host)) == NULL) {
    fprintf(stderr, "Unknown host: \%s\n", host);
    return -1;
}

saddr.sin_family = hp->h_addrtype;
memcpy((char *)&saddr.sin_addr, hp->h_addr, sizeof(saddr.sin_addr));
saddr.sin_port = htons(port);

if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("creating socket");
    return -1;
}

if (connect(s, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
    perror("connecting to server");
    (void) close(s);
    return -1;
}

return s;
```

Establishing a Security Context With the Server

After the connection is made, call_server() uses the function client_establish_context() to create the security context, as follows:

```c
if (client_establish_context(s, service-name, deleg-flag, oid, &context, &ret-flags) < 0) {
    (void) close(s);
    return -1;
}
```

- `s` is a file descriptor that represents the connection that is established by connect_to_server().
- `service-name` is the requested network service.
- `deleg-flag` specifies whether the server can act as a proxy for the client.
- `oid` is the mechanism.
- `context` is the context to be created.
- `ret-flags` is an int that specifies any flags to be returned by the GSS-API function gss_init_sec_context().

The client_establish_context() performs the following tasks:

- Translates the service name into internal GSSAPI format
Performs a loop of token exchanges between the client and the server until the security context is complete.

Translating a Service Name into GSS-API Format

The first task that `client_establish_context()` performs is to translate the service name string to internal GSS-API format by using `gss_import_name()`.

**EXAMPLE 5-3 client_establish_context() – Translate Service Name**

```c
/*
 * Import the name into target_name. Use send_tok to save
 * local variable space.
 */

send_tok.value = service_name;
send_tok.length = strlen(service_name) + 1;
maj_stat = gss_import_name(&min_stat, &send_tok,
    (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("parsing name", maj_stat, min_stat);
    return -1;
}
```

`gss_import_name()` takes the name of the service, which is stored in an opaque GSS_API buffer `send_tok`, and converts the string to the GSS_API internal name `target_name`. `send_tok` is used to save space instead of declaring a new `gss_buffer_desc`. The third argument is a `gss_OID` type that indicates the `send_tok` name format. This example uses `GSS_C_NT_HOSTBASED_SERVICE`, which means a service of the format `service@host`. See "Name Types" on page 222 for other possible values for this argument.

Establishing a Security Context for GSS-API

Once the service has been translated to GSS-API internal format, the context can be established. To maximize portability, establishing context should always be performed as a loop.

Before entering the loop, `client_establish_context()` initializes the context and the `token_ptr` parameter. There is a choice in the use of `token_ptr`. `token_ptr` can point either to `send_tok`, the token to be sent to the server, or to `recv_tok`, the token that is sent back by the server.

Inside the loop, two items are checked:

- The status that is returned by `gss_init_sec_context()`

  The return status catches any errors that might require the loop to be aborted. `gss_init_sec_context()` returns `GSS_S_CONTINUE_NEEDED` if and only if the server has another token to send.
The size of token to be sent to the server, which is generated by `gss_init_sec_context()`. A token size of zero indicates that no more information exists that can be sent to the server and that the loop can be exited. The token size is determined from `token_ptr`.

The following pseudocode describes the loop:

```c
do
  gss_init_sec_context()
  if no context was created
    exit with error;
  if the status is neither "complete" nor "in process"
    release the service namespace and exit with error;
  if there is a token to send to the server, that is, the size is nonzero
    send the token;
    if sending the token fails,
      release the token and service namespaces. Exit with error;
      release the namespace for the token that was just sent;
  if the context is not completely set up
    receive a token from the server;
while the context is not complete
```

The loop starts with a call to `gss_init_sec_context()`, which takes the following arguments:

- The status code to be set by the underlying mechanism.
- The credential handle. The example uses `GSS_C_NO_CREDENTIAL` to act as a default principal.
- The context handle to be created.
- The name of the context acceptor.
- The object ID of the desired mechanism.
- Request flags. In this case, the client requests that the server authenticate itself, that message-duplication be turned on, and that the server act as a proxy if requested.
- No time limit for the context.
- No request for channel bindings.
- Token to be received from the peer application.
- The mechanism actually used by the server. The mechanism is set to `NULL` here because the application does not use this value.
- The token to send to the peer application, which is the token that `gss_init_sec_context()` creates.
- Return flags. Set to `NULL` because they are ignored in this example.
**Note** – The client does not need to acquire credentials before initiating a context. On the client side, credential management is handled transparently by the GSS-API. That is, the GSS-API knows how to get credentials that are created by this mechanism for this principal. As a result, the application can pass `gss_init_sec_context()` a default credential. On the server side, however, a server application must explicitly acquire credentials for a service before accepting a context. See “Acquiring Credentials” on page 110.

After checking that a context or part of one exists and that `gss_init_sec_context()` is returning valid status, `connect_to_server()` checks that `gss_init_sec_context()` has provided a token to send to the server. If no token is present, the server has signalled that no other tokens are needed. If a token has been provided, then that token must be sent to the server. If sending the token fails, the namespaces for the token and service cannot be determined, and `connect_to_server()` exits. The following algorithm checks for the presence of a token by looking at the length:

```c
if (send_tok_length != 0) {
    if (send_token(s, &send_tok) < 0) {
        (void) gss_release_buffer(&min_stat, &send_tok);
        (void) gss_release_name(&min_stat, &target_name);
        return -1;
    }
}
```

`send_token()` is not a GSS-API function and needs to be written by the user. The `send_token()` function writes a token to the file descriptor. `send_token()` returns 0 on success and -1 on failure. GSS-API does not send or receive tokens itself. The calling applications are responsible for sending and receiving any tokens that have been created by GSS-API.

The source code for the context establishment loop is provided below.

**EXAMPLE 5-4**  Loop for Establishing Contexts

```c
/*
 * Perform the context establishment loop.
 * On each pass through the loop, token_ptr points to the token to send to the server (or GSS_C_NO_BUFFER on the first pass).
 * Every generated token is stored in send_tok which is then transmitted to the server; every received token is stored in recv_tok, which token_ptr is then set to, to be processed by the next call to gss_init_sec_context.
 * GSS-API guarantees that send_tok's length will be non-zero if and only if the server is expecting another token from us, and that gss_init_sec_context returns GSS_S_CONTINUE_NEEDED if and only if the server has another token to send us.
*/

token_ptr = GSS_C_NO_BUFFER;
gss_context = GSS_C_NO_CONTEXT;
```
do {
    maj_stat =
        gss_init_sec_context(&min_stat, GSS_C_NO_CREDENTIAL,
                            gss_context, target_name, oid,
                            GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG | deleg_flag,
                            0, NULL, 
                            /* no channel bindings */
                            token_ptr, NULL, 
                            /* ignore mech type */
                            &send_tok, ret_flags, NULL); /* ignore time_rec */
    if (gss_context == NULL){
        printf("Cannot create context\n");
        return GSS_S_NO_CONTEXT;
    }
    if (token_ptr != GSS_C_NO_BUFFER)
        (void) gss_release_buffer(&min_stat, &recv_tok);
    if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
        display_status("initializing context", maj_stat, min_stat);
        (void) gss_release_name(&min_stat, &target_name);
        return -1;
    }
    if (send_tok.length != 0){
        fprintf(stdout, "Sending init_sec_context token (size=%ld)...",
                send_tok.length);
        if (send_token(s, &send_tok) < 0) {
            (void) gss_release_buffer(&min_stat, &send_tok);
            (void) gss_release_name(&min_stat, &target_name);
            return -1;
        }
    }
    (void) gss_release_buffer(&min_stat, &send_tok);
}

for more information about how send_token() and recv_token() work, see “Miscellaneous
GSS-API Sample Functions” on page 207.
Miscellaneous GSSAPI Context Operations on the Client Side

As a sample program, gss-client performs some functions for demonstration purposes. The following source code is not essential for the basic task, but is provided to demonstrate these other operations:

- Saving and restoring the context
- Displaying context flags
- Obtaining the context status

The source code for these operations is shown in the following example.

```
EXAMPLE 5-5  gss-client: call_server() Establish Context
/* Save and then restore the context */
maj_stat = gss_export_sec_context(&min_stat,
&context,
&context_token);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("exporting context", maj_stat, min_stat);
    return -1;
}
maj_stat = gss_import_sec_context(&min_stat,
&context_token,
&context);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("importing context", maj_stat, min_stat);
    return -1;
}
(void) gss_release_buffer(&min_stat, &context_token);

/* display the flags */
display_ctx_flags(ret_flags);

/* Get context information */
maj_stat = gss_inquire_context(&min_stat, context,
&src_name, &targ_name, &lifetime,
&mechanism, &context_flags, 
&is_local, 
&is_open);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("inquiring context", maj_stat, min_stat);
    return -1;
}
if (maj_stat == GSS_S_CONTEXT_EXPIRED) {
    printf("context expired
");
    display_status("Context is expired", maj_stat, min_stat);
    return -1;
}
```
Wrapping and Sending a Message

The gss-client application needs to wrap, that is, encrypt the data before the data can be sent. The application goes through the following steps to wrap the message:

- Determines the wrap size limit. This process ensures that the wrapped message can be accommodated by the protocol.
- Obtains the source and destination names. Translates the names from object identifiers to strings.
- Gets the list of mechanism names. Translates the names from object identifiers to strings.
- Inserts the message into a buffer and wraps the message.
- Sends the message to the server.

The following source code wraps a message.

EXAMPLE 5-6  gss-client Example: callserver() – Wrap Message

```c
/* Test gss_wrap_size_limit */
maj_stat = gss_wrap_size_limit(&min_stat, context, conf_req_flag,
    GSS_C_QOP_DEFAULT, req_output_size, &max_input_size);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("wrap_size_limit call", maj_stat, min_stat);
} else
    fprintf (stderr,
"gss_wrap_size_limit returned
max input size = %d \n\n"for req_output_size = %d with Integrity only\n\nmax_input_size , req_output_size , conf_req_flag);
conf_req_flag = 1;
maj_stat = gss_wrap_size_limit(&min_stat, context, conf_req_flag,
    GSS_C_QOP_DEFAULT, req_output_size, &max_input_size);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("wrap_size_limit call", maj_stat, min_stat);
} else
    fprintf (stderr,
"gss_wrap_size_limit returned
max input size = %d \n\n"for req_output_size = %d with Integrtiy & Privacy\n\nmax_input_size , req_output_size 
max_input_size , req_output_size , conf_req_flag);

maj_stat = gss_display_name(&min_stat, src_name, &sname, &name_type);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying source name", maj_stat, min_stat);
    return -1;
}

maj_stat = gss_display_name(&min_stat, targ_name, &tname,
                        (gss_OID *) NULL);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying target name", maj_stat, min_stat);
    return -1;
}

fprintf(stderr,
"\%.*s\" to \%.*s\", lifetime %u, flags %x, %s, %s\n",
(int) sname.length, (char *) sname.value, (int) tname.length,
(char *) tname.value, lifetime, context_flags,
```
EXAMPLE 5–6  gss-client Example: call_server() – Wrap Message  (Continued)

(is local) ? "locally initiated",
(is_open) ? "open": "closed");

(void) gss_release_name(&min_stat, &src_name);
(void) gss_release_name(&min_stat, &targ_name);
(void) gss_release_buffer(&min_stat, &sname);
(void) gss_release_buffer(&min_stat, &tname);

maj_stat = gss_oid_to_str(&min_stat, name_type, &oid_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("converting oid->string", maj_stat, min_stat);
    return -1;
}

(void) gss_release_buffer(&min_stat, &oid_name);
/* Now get the names supported by the mechanism */
maj_stat = gss_inquire_names_for_mech(&min_stat, mechanism, &mech_names);
if (maj_stat != GSS_S COMPLETE) {
    display_status("inquiring mech names", maj_stat, min_stat);
    return -1;
}

maj_stat = gss_oid_to_str(&min_stat, mechanism, &oid_name);
if (maj_stat != GSS_S COMPLETE) {
    display_status("converting oid->string", maj_stat, min_stat);
    return -1;
}

mechStr = (char *)__gss_oid_to_mech(mechanism);
(void) gss_release_buffer(&min_stat, &oid_name);

for (i=0; i < mech_names->count; i++) {
    maj_stat = gss_oid_to_str(&min_stat, &mech_names->elements[i], &oid_name);
    if (maj_stat != GSS_S COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    printf(stderr, "%d: %.*s
", i, (int) oid_name.length, (char *) oid_name.value);
    (void) gss_release_buffer(&min_stat, &oid_name);
}
(void) gss_release_oid_set(&min_stat, &mech_names);

if (use_file) {
    read_file(msg, &in_buf);
} else {
    /* Wrap the message */
    in_buf.value = msg;
    in_buf.length = strlen(msg) + 1;
}
EXAMPLE 5–6  gss-client Example: call_server() – Wrap Message  (Continued)

if (ret_flag & GSS_C_CONF_FLAG) {
    state = 1;
} else {
    state = 0;
}
maj_stat = gss_wrap(&min_stat, context, 1, GSS_C_QOP_DEFAULT, &in_buf, &state, &out_buf);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("wrapping message", maj_stat, min_stat);
    (void) close(s);
    (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
    return -1;
} else if (! state) {
    fprintf(stderr, "Warning! Message not encrypted.\n");
}
/* Send to server */
if (send_token(s, &out_buf) < 0) {
    (void) close(s);
    (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
    return -1;
}
(void) gss_release_buffer(&min_stat, &out_buf);

Reading and Verifying a Signature Block From a GSS-API Client

The gss-client program can now test the validity of the message that was sent. The server returns the MIC for the message that was sent. The message can be retrieved with the recv_token().

The gss_verify_mic() function is then used to verify the message’s signature, that is, the MIC. gss_verify_mic() compares the MIC that was received with the original, unwrapped message. The received MIC comes from the server’s token, which is stored in out_buf. The MIC from the unwrapped version of the message is held in in_buf. If the two MICs match, the message is verified. The client then releases the buffer for the received token, out_buf.

The process of reading and verifying a signature block is demonstrated in the following source code.

EXAMPLE 5–7  gss-client Example – Read and Verify Signature Block

/* Read signature block into out_buf */
if (recv_token(s, &out_buf) < 0) {
    (void) close(s);
    (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
    return -1;
}
/* Verify signature block */
    maj_stat = gss_(&min_stat, context, &in_buf,
                  &out_buf, &qop_state);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("verifying signature", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);
    if (use_file)
        free(in_buf.value);
    printf("Signature verified.\n");

Deleting the Security Context

The call_server() function finishes by deleting the context and returning to the main() function.

EXAMPLE 5-8  gss-client Example: call_server() – Delete Context

/* Delete context */
    maj_stat = gss_delete_sec_context(&min_stat, &context, &out_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("deleting context", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);
    (void) close(s);
    return 0;
CHAPTER 6

GSS-API Server Example

This chapter presents a walk-through of the source code for the gss-server sample program. The following topics are covered:

- “GSSAPI Server Example Overview” on page 107
- “GSSAPI Server Example: main() Function” on page 108
- “Acquiring Credentials” on page 110
- “Checking for inetd” on page 113
- “Receiving Data From a Client” on page 113
- “Cleanup in the GSSAPI Server Example” on page 121

GSSAPI Server Example Overview

The sample server-side program gss-server works in conjunction with gss-client, which is described in the previous chapter. The basic purpose of gss-server is to receive, sign, and return the wrapped message from gssapi-client.

The following sections provide a step-by-step description of how gss-server works. Because gss-server is a sample program for demonstrating GSSAPI functionality, only relevant parts of the program are discussed in detail. The complete source code for the two applications appears in the appendix and can be downloaded from

GSSAPI Server Example Structure

The gss-structure application performs the following steps:

1. Parses the command line.
2. If a mechanism is specified, translates the mechanism name to internal format.
3. Acquires credentials for the caller.
4. Checks to see whether the user has specified using the inetd daemon for connecting.
5. Makes a connection with the client.
6. Receives the data from the client.
7. Signs and returns the data.
8. Releases namespaces and exits.

**Running the GSSAPI Server Example**

gss-server takes this form on the command line

```
[-mech mechanism] service-name
```

- `port` is the port number to listen on. If no port is specified, the program uses port 4444 as the default.
- `-verbose` causes messages to be displayed as gss-server runs.
- `-inetd` indicates that the program should use the inetd daemon to listen to a port. `-inetd` uses stdin and stdout to connect to the client.
- `-once` indicates a single-instance connection only.
- `mechanism` is the name of a security mechanism to use, such as Kerberos v5. If no mechanism is specified, the GSS-API uses a default mechanism.
- `service-name` is the name of the network service that is requested by the client, such as telnet, ftp, or login service.

A typical command line might look like the following example:

```
% gss-server -port 8080 -once -mech kerberos_v5 erebos.eng nfs "hello"
```

**GSSAPI Server Example: main() Function**

The `gss-server main()` function performs the following tasks:

- Parses command-line arguments and assigns the arguments to variables
- Acquires the credentials for the service corresponding to the mechanism
- Calls the `sign_server()` function, which performs the work involved with signing and returning the message
- Releases the credentials that have been acquired
- Releases the mechanism OID namespace
- Closes the connection if the connection is still open
EXAMPLE 6-1  gss-server Example: main()

int main(argc, argv)
    int argc;
    char **argv;
{
    char *service_name;
    gss_cred_id_t server_creds;
    OM_uint32 min_stat;
    u_short port = 4444;
    int s;
    int once = 0;
    int do_inetd = 0;

    log = stdout;
    display_file = stdout;

    /* Parse command-line arguments. */
    argc--; argv++;
    while (argc) {
        if (strcmp(*argv, "-port") == 0) {
            argc--; argv++;
            if (!argc) usage();
            port = atoi(*argv);
        } else if (strcmp(*argv, "-verbose") == 0) {
            verbose = 1;
        } else if (strcmp(*argv, "-once") == 0) {
            once = 1;
        } else if (strcmp(*argv, "-inetd") == 0) {
            do_inetd = 1;
        } else if (strcmp(*argv, "-logfile") == 0) {
            argc--; argv++;
            if (!argc) usage();
            log = fopen(*argv, "a");
            display_file = log;
            if (!log) {
                perror(*argv);
                exit(1);
            }
        } else
            break;
    argc--; argv++;
}
    if (argc != 1)
        usage();

    if (argv[0][0] == '-')
        usage();

    service_name = *argv;

    /* Acquire service credentials. */
    if (server_acquire_creds(service_name, &server_creds) < 0)
        return -1;

    if (do_inetd) {
        close(1);
        close(2);
Acquiring Credentials

Credentials are created by the underlying mechanisms rather than by the client application, server application, or GSS-API. A client program often has credentials that are obtained at login. A server always needs to acquire credentials explicitly.

The gss-server program has a function, server_acquire_creds(), to get the credentials for the service to be provided. server_acquire_creds() takes as input the name of the service and the security mechanism to be used. server_acquire_creds() then returns the credentials for the service.

server_acquire_creds() uses the GSS-API function gss_acquire_cred() to get the credentials for the service that the server provides. Before server_acquire_creds() accesses gss_acquire_cred(), server_acquire_creds() must take care of two tasks:

1. Checking for a list of mechanisms and reducing the list to a single mechanism for the purpose of getting a credential.
If a single credential can be shared by multiple mechanisms, the `gss_acquire_cred()` function returns credentials for all those mechanisms. Therefore, `gss_acquire_cred()` takes as input a set of mechanisms. (See “Working With Credentials in GSS-API” on page 72.) In most cases, however, including this one, a single credential might not work for multiple mechanisms. In the `gss-server` program, either a single mechanism is specified on the command line or else the default mechanism is used. Therefore, the first task is to make sure that the set of mechanisms that was passed to `gss_acquire_cred()` contains a single mechanism, default or otherwise, as follows:

```c
if (mechOid != GSS_C_NULL_OID) {
    desiredMechs = &mechOidSet;
    mechOidSet.count = 1;
    mechOidSet.elements = mechOid;
} else
    desiredMechs = GSS_C_NULL_OID_SET;
```

`GSS_C_NULL_OID_SET` indicates that the default mechanism should be used.

2. Translating the service name into GSS-API format.

Because `gss_acquire_cred()` takes the service name in the form of a `gss_name_t` structure, the name of the service must be imported into that format. The `gss_import_name()` function performs this translation. Because this function, like all GSS-API functions, requires arguments to be GSS-API types, the service name has to be copied to a GSS-API buffer first, as follows:

```c
name_buf.value = service_name;
name_buf.length = strlen(name_buf.value) + 1;
min_stat = gss_import_name(&min_stat, &name_buf,
                           (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("importing name", maj_stat, min_stat);
    if (mechOid != GSS_C_NO_OID)
        gss_release_oid(&min_stat, &mechOid);
    return -1;
}
```

Note again the use of the nonstandard function `gss_release_oid()`.

The input is the service name as a string in `name_buf`. The output is the pointer to a `gss_name_t` structure, `server_name`. The third argument, `GSS_C_NT_HOSTBASED_SERVICE`, is the name type for the string in `name_buf`. In this case, the name type indicates that the string should be interpreted as a service of the format `service@host`.

After these tasks have been performed, the server program can call `gss_acquire_cred()`:

```c
maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                            desiredMechs, GSS_C_ACCEPT,
                            server_creds, NULL, NULL);
```

- `min_stat` is the error code returned by the function.
- `server_name` is the name of the server.
- `0` indicates that the program does not need to know the maximum lifetime of the credential.
- **desiredMechs** is the set of mechanisms for which this credential applies.
- GSS_C_ACCEPT means that the credential can be used only to accept security contexts.
- **server_creds** is the credential handle to be returned by the function.
- NULL, NULL indicates that the program does not need to know either the specific mechanism being employed or the amount of time that the credential will be valid.

The following source code illustrates the `server_acquire_creds()` function.

**EXAMPLE 6-2  Sample Code for `server_acquire_creds()` Function**

```c
/*
 * Function: server_acquire_creds
 * Purpose: imports a service name and acquires credentials for it
 * 
 * Arguments:
 * 
 * service_name (r) the ASCII service name
 * mechType (r) the mechanism type to use
 * server_creds (w) the GSS-API service credentials
 * 
 * Returns: 0 on success, -1 on failure
 * 
 * Effects:
 * 
 * The service name is imported with gss_import_name, and service
 * credentials are acquired with gss_acquire_cred. If either operation
 * fails, an error message is displayed and -1 is returned; otherwise,
 * 0 is returned.
 */

int server_acquire_creds(service_name, mechOid, server_creds)
    char *service_name;
    gss_OID mechOid;
    gss_cred_id_t *server_creds;
{
    gss_buffer_desc name_buf;
    gss_name_t server_name;
    OM_uint32 maj_stat, min_stat;
    gss_OID_set desc mechOidSet;
    gss_OID_set desiredMechs = GSS_C_NULL_OID_SET;

    if (mechOid != GSS_C_NULL_OID) {
        desiredMechs = &mechOidSet;
        mechOidSet.count = 1;
        mechOidSet.elements = mechOid;
    } else
        desiredMechs = GSS_C_NULL_OID_SET;

    name_buf.value = service_name;
    name_buf.length = strlen(name_buf.value) + 1;
    maj_stat = gss_import_name(&min_stat, &name_buf,
            (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("importing name", maj_stat, min_stat);
        if (mechOid != GSS_C_NO_OID)
```
EXAMPLE 6-2  Sample Code for server_acquire_creds() Function  (Continued)

         gss_release_oid(&min_stat, &mechOid);
         return -1;
     }

     maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
         desiredMechs, GSS_C_ACCEPT,
         server_creds, NULL, NULL);

     if (maj_stat != GSS_S_COMPLETE) {
         display_status("acquiring credentials", maj_stat, min_stat);
         return -1;
     }

     (void) gss_release_name(&min_stat, &server_name);
     return 0;

Checking for inetd

Having acquired credentials for the service, gss-server checks to see whether the user has specified inetd. The main function checks for inetd as follows:

if (do_inetd) {
    close(1);
    close(2);

    If the user has specified to use inetd, then the program closes the standard output and standard error. gss-server then calls sign_server() on the standard input, which inetd uses to pass connections. Otherwise, gss-server creates a socket, accepts the connection for that socket with the TCP function accept(), and calls sign_server() on the file descriptor that is returned by accept().

    If inetd is not used, the program creates connections and contexts until the program is terminated. However, if the user has specified the -once option, the loop terminates after the first connection.

Receiving Data From a Client

After checking for inetd, the gss-server program then calls sign_server(), which does the main work of the program. sign_server() first establishes the context by calling server_establish_context().

sign_server() performs the following tasks:

- Accepts the context
Unwraps the data
Signs the data
Returns the data

These tasks are described in the subsequent sections. The following source code illustrates the
sign_server() function.

EXAMPLE 6-3  sign_server() Function

```c
int sign_server(s, server_creds)
    int s;
    gss_cred_id_t server_creds;
{
    gss_buffer_desc client_name, xmit_buf, msg_buf;
    gss_ctx_id_t context;
    OM uint32 maj_stat, min_stat;
    int i, conf_state, ret_flags;
    char *cp;

    /* Establish a context with the client */
    if (server_establish_context(s, server_creds, &context,
        &client_name, &ret_flags) < 0)
        return(-1);
    printf("Accepted connection: \%.*s\n",
           (int) client_name.length, (char *) client_name.value);
    (void) gss_release_buffer(&min_stat, &client_name);
    for (i=0; i<3 ; i++)
        if (test_import_export_context(&context))
            return -1;

    /* Receive the sealed message token */
    if (recv_token(s, &xmit_buf) < 0)
        return(-1);
    if (verbose && log) {
        fprintf(log, "Sealed message token:\n");
        print_token(&xmit_buf);
    }

    maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
        &conf_state, (gss_qop_t *) NULL);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("unsealing message", maj_stat, min_stat);
        return(-1);
    } else if (! conf_state) {
        fprintf(stderr, "Warning! Message not encrypted.\n");
    }
    (void) gss_release_buffer(&min_stat, &xmit_buf);
    fprintf(log, "Received message: ");
    cp = msg_buf.value;
    if ((isprint(cp[0]) || isspace(cp[0])) &
        (isprint(cp[1]) || isspace(cp[1]))) {
        fprintf(log, "\%.*s\n", msg_buf.length, msg_buf.value);
    }
```

Receiving Data From a Client

Developer's Guide to Oracle Solaris 10 Security  •  September 2013
EXAMPLE 6–3  sign_server() Function  (Continued)

    } else {
        printf("\n");
        print_token(&msg_buf);
    }

    /* Produce a signature block for the message */
    maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT, 
                          &msg_buf, &xmit_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("signing message", maj_stat, min_stat);
        return(-1);
    }
    (void) gss_release_buffer(&min_stat, &msg_buf);

    /* Send the signature block to the client */
    if (send_token(s, &xmit_buf) < 0)
        return(-1);
    (void) gss_release_buffer(&min_stat, &xmit_buf);

    /* Delete context */
    maj_stat = gss_delete_sec_context(&min_stat, &context, NULL);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("deleting context", maj_stat, min_stat);
        return(-1);
    }
    fflush(log);
    return(0);
}

Accepting a Context

Establishing a context typically involves a series of token exchanges between the client and the server. Both context acceptance and context initialization should be performed in loops to maintain program portability. The loop for accepting a context is very similar to the loop for establishing a context, although in reverse. Compare with "Establishing a Security Context With the Server" on page 97.

The following source code illustrates the server_establish_context() function.

EXAMPLE 6–4  server_establish_context() Function

    /* Function: server_establish_context
     * Purpose: establishes a GSS-API context as a specified service with
     * an incoming client, and returns the context handle and associated
     * client name
EXAMPLE 6-4 server_establish_context() Function  (Continued)

* Arguments:
  * s (r) an established TCP connection to the client
  * service_creds (r) server credentials, from gss_acquire_cred
  * context (w) the established GSS-API context
  * client_name (w) the client’s ASCII name

* Returns: 0 on success, -1 on failure

* Effects:
  * Any valid client request is accepted. If a context is established, its handle is returned in context and the client name is returned in client_name and 0 is returned. If unsuccessful, an error message is displayed and -1 is returned.

*/

int server_establish_context(s, server_creds, context, client_name, ret_flags)
{

  gss_buffer_desc send_tok, recv_tok;
  gss_name_t client;
  gss_OID doid;
  OM_uint32 maj_stat, min_stat, acc_sec_min_stat;
  gss_buffer_desc oid_name;

  *context = GSS_C_NO_CONTEXT;

  do {
    if (recv_token(s, &recv_tok) < 0)
      return -1;

    if (verbose && log) {
      fprintf(log, "Received token (size=%d): \n", recv_tok.length);
      print_token(&recv_tok);
    }

    maj_stat =
      gss_accept_sec_context(&acc_sec_min_stat,
                              context,
                              server_creds,
                              &recv_tok,
                              GSS_C_NO_CHANNEL_BINDINGS,
                              &client,
                              &doid,
                              &send_tok,
                              ret_flags,
                              NULL, /* ignore time_rec */
                              NULL); /* ignore del_cred_handle */

    (void) gss_release_buffer(&min_stat, &recv_tok);
  }
if (send_tok.length != 0) {
    if (verbose && log) {
        fprintf(log, "Sending accept_sec_context token (size=%d):\n", send_tok.length);
        print_token(&send_tok);
    }
    if (send_token(s, &send_tok) < 0) {
        fprintf(log, "failure sending token\n");
        return -1;
    }
    (void) gss_release_buffer(&min_stat, &send_tok);
} 
if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
    display_status("accepting context", maj_stat, acc_sec_min_stat);
    if (*context == GSS_C_NO_CONTEXT)
        gss_delete_sec_context(&min_stat, context, GSS_C_NO_BUFFER);
    return -1;
}
if (verbose && log) {
    if (maj_stat == GSS_S_CONTINUE_NEEDED)
        fprintf(log, "continue needed...\n");
    else
        fprintf(log, "\n");
    fflush(log);
} 
} 
while (maj_stat == GSS_S_CONTINUE_NEEDED);

/* display the flags */
display_ctx_flags(*ret_flags);

if (verbose && log) {
    maj_stat = gss_oid_to_str(&min_stat, doid, &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    fprintf(log, "Accepted connection using mechanism OID %.*s.\n", (int) oid_name.length, (char *) oid_name.value);
    (void) gss_release_buffer(&min_stat, &oid_name);
}

maj_stat = gss_display_name(&min_stat, client, client_name, &doid);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying name", maj_stat, min_stat);
    return -1;
}

maj_stat = gss_release_name(&min_stat, &client);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("releasing name", maj_stat, min_stat);
    return -1;
}
EXAMPLE 6–4  server_establish_context() Function  (Continued)

    return 0;
}

The sign_server() function uses the following source code to call server_establish_context() to accept the context.

    /* Establish a context with the client */
    if (server_establish_context(s, server_creds, &context,
                                &client_name, &ret_flags) < 0)
        return(-1);

The server_establish_context() function first looks for a token that the client sends as part of the context initialization process. Because, GSS-API does not send or receive tokens itself, programs must have their own routines for performing these tasks. The server uses recv_token() for receiving the token:

    do {
        if (recv_token(s, &recv_tok) < 0)
            return -1;
    }

Next, server_establish_context() calls the GSS-API function gss_accept_sec_context():

    maj_stat = gss_accept_sec_context(&min_stat,
                                       context,
                                       server_creds,
                                       &recv_tok,
                                       GSS_C_NO_CHANNEL_BINDINGS,
                                       &client,
                                       &oid,
                                       &send_tok,
                                       ret_flags,
                                       NULL, /* ignore time rec */
                                       NULL); /* ignore del_cred_handle */

- **min_stat** is the error status returned by the underlying mechanism.
- **context** is the context being established.
- **server_creds** is the credential for the service to be provided (see “Acquiring Credentials” on page 110).
- **recv_tok** is the token received from the client by recv_token().
- **GSS_C_NO_CHANNEL_BINDINGS** is a flag indicating not to use channel bindings (see “Using Channel Bindings in GSS-API” on page 79).
- **client** is the ASCII name of the client.
- **oid** is the mechanism (in OID format).
- **send_tok** is the token to send to the client.
- **ret_flags** are various flags indicating whether the context supports a given option, such as message-sequence-detection.
The two NULL arguments indicate that the program does not need to know the length of
time that the context will be valid, or whether the server can act as a client’s proxy.

The acceptance loop continues, barring any errors, as long as gss_accept_sec_context() sets
maj_stat to GSS_S_CONTINUE_NEEDED. If maj_stat is not equal to that value or to
GSS_S_COMPLETE, a problem exists and the loop exits.

gss_accept_sec_context() returns a positive value for the length of send_tok whether a token
exists to send back to the client. The next step is to see a token exists to be sent, and, if so, to send
the token:

```c
if (send_tok.length != 0) {
  ...
  if (send_token(s, &send_tok) < 0) {
    fprintf(log, "failure sending token\n");
    return -1;
  }
  (void) gss_release_buffer(&min_stat, &send_tok);
}
```

**Unwrapping the Message**

After accepting the context, the sign_server() receives the message that has been sent by the
client. Because the GSS-API does not provide a function for receiving tokens, the program uses
the recv_token() function:

```c
if (recv_token(s, &xmit_buf) < 0)
  return(-1);
```

Because the message might be encrypted, the program uses the GSS-API function
gss_unwrap() for unwrapping:

```c
maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
  &conf_state, (gss_qop_t *) NULL);
  if (maj_stat != GSS_S_COMPLETE) {
    display_status("unwrapping message", maj_stat, min_stat);
    return(-1);
  } else if (! conf_state) {
    fprintf(stderr, "Warning! Message not encrypted.\n");
  }
  (void) gss_release_buffer(&min_stat, &xmit_buf);
```

gss_unwrap() takes the message that recv_token() has placed in xmit_buf, translates the
message, and puts the result in msg_buf. Two arguments to gss_unwrap() are noteworthy.  
conf_state is a flag to indicate whether confidentiality, that is, encryption, has been applied to
this message. The final NULL indicates that the program does not need to know that the QOP
that was used to protect the message.
Signing and Returning the Message

At this point, the `sign_server()` function needs to sign the message. Signing a message entails returning the message's Message Integrity Code or MIC to the client. Returning the message proves that the message was sent and was unwrapped successfully. To obtain the MIC, `sign_server()` uses the function `gss_get_mic()`:

```c
maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT, &msg_buf, &xmit_buf);
```

gss_get_mic() looks at the message in `msg_buf`, produces the MIC, and stores the MIC in `xmit_buf`. The server then sends the MIC back to the client with `send_token()`. The client verifies the MIC with `gss_verify_mic()`. See “Reading and Verifying a Signature Block From a GSS-APIClient” on page 105.

Finally, `sign_server()` performs some cleanup. `sign_server()` releases the GSS-API buffers `msg_buf` and `xmit_buf` with `gss_release_buffer()`. Then `sign_server()` destroys the context with `gss_delete_sec_context()`.

Using the `test_import_export_context()` Function

GSS-API allows you to export and import contexts. These activities enable you to share a context between different processes in a multiprocess program. `sign_server()` contains a proof-of-concept function, `test_import_export_context()`, that illustrates how exporting and importing contexts works. `test_import_export_context()` does not pass a context between processes. Instead, `test_import_export_context()` displays the amount of time to export and then import a context. Although an artificial function, `test_import_export_context()` does indicate how to use the GSS-API importing and exporting functions. `test_import_export_context()` also shows how to use timestamps with regard to manipulating contexts.

The source code for `test_import_export_context()` is shown in the following example.

```c
EXAMPLE 6-5  test_import_export_context()
int test_import_export_context(context)
    gss_ctx_id_t *context;
{
    OM_uint32       min_stat, maj_stat;
    gss_buffer_desc context_token, copied_token;
    struct timeval tm1, tm2;

    /* Attempt to save and then restore the context. */
    gettimeofday(&tm1, (struct timezone *)0);
    maj_stat = gss_export_sec_context(&min_stat, context, &context_token);
    if (maj_stat != GSS_S_COMPLETE) {
    ```
EXAMPLE 6-5  \( \text{test\_import\_export\_context}() \)  

(Continued)

```c
    display_status("exporting context", maj_stat, min_stat);
    return 1;
}
```

```c
    gettimeofday(&tm2, (struct timezone *)0);
    if (verbose && log)
        fprintf(log, "Exported context: %d bytes, %7.4f seconds\n",
                context_token.length, timeval_subtract(&tm2, &tm1));
    copied_token.length = context_token.length;
    copied_token.value = malloc(context_token.length);
    if (copied_token.value == 0) {
        fprintf(log, "Couldn’t allocate memory to copy context token.\n"
                );
        return 1;
    }
    memcpy(copied_token.value, context_token.value, copied_token.length);
    maj_stat = gss_import_sec_context(&min_stat, &copied_token, context);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("importing context", maj_stat, min_stat);
        return 1;
    }
    free(copied_token.value);
    gettimeofday(&tm1, (struct timezone *)0);
    if (verbose && log)
        fprintf(log, "Importing context: %7.4f seconds\n",
                timeval_subtract(&tm1, &tm2));
    (void) gss_release_buffer(&min_stat, &context_token);
    return 0;
```

Cleanup in the GSSAPI Server Example

Back in the \texttt{main()} function, the application deletes the service credential with \texttt{gss\_release\_cred()}.
If an OID for the mechanism has been specified, the program deletes the OID with \texttt{gss\_release\_oid()} and exits.

```c
    (void) gss_release_cred(&min_stat, &server_creds);
```
Introduction to Simple Authentication Security Layer (SASL)

SASL provides developers of applications and shared libraries with mechanisms for authentication, data integrity-checking, and encryption. SASL enables the developer to code to a generic API. This approach avoids dependencies on specific mechanisms. SASL is particularly appropriate for applications that use the IMAP, SMTP, ACAP, and LDAP protocols, as these protocols all support SASL. SASL is described in RFC 2222.

SASL Library Basics

The SASL library is called `libsasl`. `libsasl` is a framework that allows properly written SASL consumer applications to use any SASL plug-ins that are available on the system. The term `plug-in` refers to objects that provide services for SASL. Plug-ins are external to `libsasl`. SASL plug-ins can be used for authentication and security, canonicalization of names, and lookup of auxiliary properties, such as passwords. Cryptographic algorithms are stored in plug-ins rather than in `libsasl`.

`libsasl` provides an application programming interface (API) for consumer applications and libraries. A service provider interface (SPI) is provided for plug-ins to supply services to `libsasl`. `libsasl` is not aware of the network or the protocol. Accordingly, the application must take responsibility for sending and receiving data between the client and server.
SASL uses two important identifiers for users. The *authentication ID* \((\text{authid})\) is the user ID for authenticating the user. The authentication ID grants the user access to a system. The *authorization ID* \((\text{userid})\) is used to check whether the user is allowed to use a particular option.

The SASL client application and SASL server application negotiate a common SASL mechanism and security level. Typically, the SASL server application sends its list of acceptable authentication mechanisms to the client. The SASL client application can then decide which authentication mechanism best satisfies its requirements. After this point, the authentication takes place using the agreed-upon authentication mechanism as a series of client-server exchanges of the SASL supplied authentication data. This exchange continues until the authentication successfully completes, fails, or is aborted by the client or the server.

In the process of authentication, the SASL authentication mechanism can negotiate a security layer. If a security layer is selected, that layer must be used for the duration of the SASL session.

**SASL Architecture**

The following figure shows the basic SASL architecture.

![SASL Architecture Diagram](image)

Client and server applications make calls to their local copies of \libasasl\ through the SASL API. \libasasl\ communicates with the SASL mechanisms through the SASL service provider interface (SPI).

**Security Mechanisms**

Security mechanism plug-ins provide security services to \libasasl\. Some typical functions that are provided by security mechanisms follow:

- Authentication on the client side
- Authentication on the server side
- Integrity, that is, checking that transmitted data is intact
- Confidentiality, that is, encrypting and decrypting transmitted data

**SASL Security Strength Factor**

SSF, the security strength factor, indicates the strength of the SASL protection. If the mechanism supports a security layer, the client and server negotiate the SSF. The value of the SSF is based on the security properties that were specified before the SASL negotiation. If a non-zero SSF is negotiated, both client and server need to use the mechanism’s security layer when the authentication has completed. SSF is represented by an integer with one of the following values:

- 0 – No protection.
- 1 – Integrity checking only.
- >1 – Supports authentication, integrity and confidentiality. The number represents the encryption key length.

The confidentiality and integrity operations are performed by the security mechanism. `libsasl` coordinates these requests.

**Note** – In the negotiation, the SASL client selects the mechanism with the maximum SSF. However, the actual SASL mechanism that is chosen might subsequently negotiate a lower SSF.

**Communication in SASL**

Applications communicate with `libsasl` through the `libsasl` API. `libsasl` can request additional information by means of callbacks that are registered by the application. Applications do not call plug-ins directly, only through `libsasl`. Plug-ins generally call the `libsasl` framework’s plug-ins, which then call the application’s callbacks. SASL plug-ins can also call the application directly, although the application does not know whether the call came from a plug-in or from `libsasl`.

Callbacks are useful in multiple areas, as follows.

- `libsasl` can use callbacks to get information that is needed to complete authentication.
- `libsasl` consumer applications can use callbacks to change search paths for plug-ins and configuration data, to verify files, and to change various default behaviors.
- Servers can use callbacks to change authorization policies, to supply different password verification methods, and to get password change information.
- Clients and servers can use callbacks to specify the language for error messages.

Applications register two sorts of callbacks: global and session. Additionally, `libsasl` defines a number of callback identifiers that are used to register for different sorts of callbacks. If a given type of callback is not registered, `libsasl` takes default action.
Session callbacks override global callbacks. If a session callback is specified for a given ID, the global callback is not called for that session. Some callbacks must be global, because these callbacks occur outside of sessions. The following instances require global callbacks:

- Determination of search paths for plug-ins to load
- Verification of plug-ins
- Location of configuration data
- The logging of error messages
- Other global configuration of \texttt{libsasl} or its plug-ins

A SASL callback can be registered with a NULL callback function for a given SASL callback ID. The NULL callback function indicates that the client is equipped to supply the needed data. All SASL callback IDs start with the prefix \texttt{SASL\_CB\_}.

SASL provides the following callbacks for use by either a client or a server:

- \texttt{SASL\_CB\_GETOPT} Gets a SASL option. Options modify the behavior of \texttt{libsasl(3LIB)} and related plug-ins. Can be used by either a client or a server.

- \texttt{SASL\_CB\_LOG} Sets the logging function for \texttt{libsasl} and its plug-ins. The default behavior is to use \texttt{syslog}.

- \texttt{SASL\_CB\_GETPATH} Gets the colon-separated list of SASL plug-in search paths. The default paths depend on the architecture as follows:
  - 32-bit SPARC architecture: /usr/lib/sasl
  - 32-bit x86 architecture: /usr/lib/sasl
  - 64-bit SPARC architecture: /usr/lib/sasl/sparcv9
  - x64 architecture: /usr/lib/sasl/amd64

- \texttt{SASL\_CB\_GETCONF} Gets the path to the SASL server's configuration directory. The default is /etc/sasl.

- \texttt{SASL\_CB\_LANGUAGE} Specifies a comma-separated list of RFC 1766 language codes in order of preference, for client and server error messages and for client prompts. The default is \texttt{i-default}.

- \texttt{SASL\_CB\_VERIFYFILE} Verifies the configuration file and plug-in files.

SASL provides the following callbacks for use by clients only:

- \texttt{SASL\_CB\_USER} Gets the client user name. The user name is the same as the authorization ID. The \texttt{LOGNAME} environment variable is the default.

- \texttt{SASL\_CB\_AUTHNAME} Gets the client authentication name.

- \texttt{SASL\_CB\_PASS} Gets a client passphrase-based secret.
SASL provides the following callbacks for use by servers only:

**SASL_CB_ECHOPROMPT**  
Gets the result for a given challenge prompt. The input from the client can be echoed.

**SASL_CB_NOECHOPROMPT**  
Gets the result for a given challenge prompt. The input from the client should not be echoed.

**SASL_CB_GETREALM**  
Sets the realm to be used for authentication.

When the SASL library is first initialized, the server and client declare any necessary global callbacks. The global callbacks are available prior to and during the SASL sessions. Prior to initialization, callbacks perform such tasks as loading plug-ins, logging data, and reading configuration files. At the start of a SASL session, additional callbacks can be declared. Such callbacks can override global callbacks if necessary.

### SASL Connection Contexts

libssas uses a SASL connection *context* to maintain the state of each SASL session for both SASL clients and SASL servers. Each context can be used for only one authentication and security session at a time. The maintained state includes the following information:

- Connection information, such as service, naming and address information, and protocol flags
- Callbacks specific to the connection
- Security properties for negotiating the SASL SSF
- State of the authentication along with security layer information
Steps in the SASL Cycle

The following diagram shows steps in the SASL life cycle. The client actions are shown on the left of the diagram and the server actions on the right side. The arrows in the middle show interactions between the client and server over an external connection.

The sections that follow illustrate the steps in the life cycle.

**Libsasl Initialization**

The client calls `sasl_client_init()` to initialize `libsasl` for the client's use. The server calls `sasl_server_init()` to initialize `libsasl` for server use.
When `sasl_client_init()` is run, the SASL client, the client's mechanisms and the client's canonicalization plug-in are loaded. Similarly, when `sasl_server_init()` is called, the SASL server, the server's mechanisms, the server's canonicalization plug-in, and the server's auxprop plug-in are loaded. After `sasl_client_init()` has been called, additional client plug-ins can be added by using `sasl_client_add_plugin()` and `sasl_canonuser_add_plugin()`. On the server side, after `sasl_server_init()` has been called, additional server plug-ins can be added through `sasl_server_add_plugin()`, `sasl_canonuser_add_plugin()`, and `sasl_auxprop_add_plugin()`. SASL mechanisms are provided in the Solaris software in the following directories according to the architecture:

- 32-bit SPARC architecture: `/usr/lib/sasl`
- 32-bit x86 architecture: `/usr/lib/sasl`
- 64-bit SPARC architecture: `/usr/lib/sasl/sparcv9`
- x64 architecture: `/usr/lib/sasl/amd64`

The `SASL_CB_GETPATH` callback can be used to override the default location.

At this point, any required global callbacks are set. SASL clients and servers might include the following callbacks:

- `SASL_CB_GETOPT`
- `SASL_CB_LOG`
- `SASL_CB_GETPATH`
- `SASL_CB_VERIFYFILE`

A SASL server might additionally include the `SASL_CB_GETCONF` callback.

**SASL Session Initialization**

The server and client use establish the connection through the protocol. To use SASL for authentication, the server and client create SASL connection contexts by using `sasl_server_new()` and `sasl_client_new()` respectively. The SASL client and server can use `sasl_setprop()` to set properties that impose security restrictions on mechanisms. This approach enables a SASL consumer application to decide the minimum SSF, the maximum SSF, and the security properties for the specified SASL connection context.

```c
#define SASL_SEC_NOPALIUNTEXT 0x0001
#define SASL_SEC_NOACTIVE 0x0002
#define SASL_SEC_NODICTIONARY 0x0004
#define SASL_SEC_FORWARD_SECRECY 0x0008
#define SASL_SEC_NOANONYMOUS 0x0010
#define SASL_SEC_PASS_CREDENTIALS 0x0020
#define SASL_SEC_MUTUAL_AUTH 0x0040
```
Note – Authentication and a security layer can be provided by the client-server protocol or by some other mechanism that is external to libsasl. In such a case, sasl_setprop() can be used to set the external authentication ID or the external SSF. For example, consider the case in which the protocol uses SSL with client authentication to the server. In this case, the external authentication identity can be the client’s subject name. The external SSF can be the key size.

For the server, libsasl determines the available SASL mechanisms according to the security properties and the external SSF. The client obtains the available SASL mechanisms from the SASL server through the protocol.

For a SASL server to create a SASL connection context, the server should call sasl_server_new(). An existing SASL connection context that is no longer in use can be reused. However, the following parameters might need to be reset:

```c
#define SASL_DEFUSERREALM 3  /* default realm passed to server_new or set with setprop */
#define SASL_IPLOCALPORT 8  /* iplocalport string passed to server_new */
#define SASL_IPREMOTEPOR9 9  /* ipremoteport string passed to server_new */
#define SASL_SERVICE 12  /* service passed to sasl_*_new */
#define SASL_SERVERFQDN 13  /* serverFQDN passed to sasl_*_new */
```

You can modify any of the parameters to sasl_client_new() and sasl_server_new() except the callbacks and protocol flags.

The server and client can also establish security policy and set connection specific parameters by using sasl_setprop() to specify the following properties:

```c
#define SASL_SSF_EXTERNAL 100 /* external SSF active (sasl_ssf_t *) */
#define SASL_SEC_PROPS 101 /* sasl_security_properties_t */
#define SASL_AUTH_EXTERNAL 102 /* external authentication ID (const char *) */
```

- **SASL_SSF_EXTERNAL** – For setting the strength factor, that is, the number of bits in the key
- **SASL_SEC_PROPS** – For defining security policy
- **SASL_AUTH_EXTERNAL** – The external authentication ID

The server can call sasl_listmech() to get a list of the available SASL mechanisms that satisfy the security policy. The client can generally get the list of available mechanisms from the server in a protocol-dependent way.

The initialization of a SASL session is illustrated in the following diagram. In this diagram and subsequent diagrams, data checks after transmission over the protocol have been omitted for the sake of simplicity.
SASL Authentication

Authentication takes a variable number of client and server steps depending on the security mechanism that is used. The SASL client calls sasl_client_start() with a list of security mechanisms to use. This list typically comes from the server. libsasl selects the best mechanism to use for this SASL session, according to the available mechanisms and the client’s security policy. The client’s security policy controls which mechanisms are permitted. The selected mechanism is returned by sasl_client_start(). Sometimes the security mechanism for the client sometimes needs additional information for authentication. For registered callbacks, libsasl calls the specified callback unless the callback function is NULL. If the callback function is NULL, libsasl returns SASL_INTERACT and a request for needed information. If SASL_INTERACT is returned, then sasl_client_start() should be called with the requested information.

If sasl_client_start() returns SASL_CONTINUE or SASL_OK, the client should send the selected mechanism with any resulting authentication data to the server. If any other value is returned, an error has occurred. For example, no mechanism might be available.

The server receives the mechanism that has been selected by the client, along with any authentication data. The server then calls sasl_server_start() to initialize the mechanism
data for this session. `sasl_server_start()` also processes any authentication data. If `sasl_server_start()` returns `SASL_CONTINUE` or `SASL_OK`, the server sends authentication data. If `sasl_server_start()` returns any other value, an error has occurred such as an unacceptable mechanism or an authentication failure. The authentication must be aborted. The SASL context should be either freed or reused.

This part of the authentication process is illustrated in the following diagram.

**FIGURE 7–4** SASL Authentication: Sending Client Data

- **Client Actions**
  - Call `sasl_client_start()`
  - Call `sasl_client_step()`
  - Get requested data.
  - Send selected SASL mechanism and SASL data to server.

- **Server Actions**
  - Get SASL mechanism and client SASL data.
  - Call `sasl_server_start()`
  - Free SASL context
  - Error
  - SASL_CONTINUE
  - SASL_OK
If the server call to `sasl_server_start()` returns `SASL_CONTINUE`, the server continues to communicate with the client to get all the necessary authentication information. The number of subsequent steps depends on the mechanism. If needed, the client calls `sasl_client_step()` to process the authentication data from the server and to generate a reply. Similarly, the server can call `sasl_server_step()` to process the authentication from the client and to generate a reply in turn. This exchange continues until the authentication is complete or until an error has occurred. SASL_OK is returned to indicate that the authentication has successfully completed for the client or server. The SASL mechanism might still have additional data to send to the other side so the other side can complete authentication. When authentication has been achieved on both sides, the server and client can inquire about each other's properties.

The following diagram shows the interactions between the server and client to transfer the additional authentication data.
**FIGURE 7-5** SASL Authentication: Processing Server Data

- **Client Actions**
  - Call `sasl_client_step()`
  - Get requested data.
  - If `SASL_OK`, Last return
  - If `SASL_INTERACT`, continue
  - If error, check return
  - Free SASL context
  - Send SASL data to server.

- **Server Actions**
  - Send SASL data to client.
  - Get SASL data from client.
  - If `SASL_OK`, Last return
  - If `SASL_CONTINUE`, continue
  - If error, call `sasl_server_step()`
  - If authenticated, continue
  - Free SASL context
SASL Confidentiality and Integrity

To check for a security layer, use the `sasl_getprop()` function to see if the security strength factor (SSF) has a value that is greater than 0. If a security layer has been negotiated, the client and server must use the resulting SSF after successful authentication. Data is exchanged between the client and server in a similar fashion to authentication. `sasl_encode()` is applied to data before the data is sent by the protocol to the client or server. On the receiving end, data is decoded by `sasl_decode()`. If a security layer has not been negotiated, the SASL connection context is not needed. The context can then be disposed of or reused.

Releasing SASL Sessions

A SASL connection context should only be freed when the session is not to be reused. `sasl_dispose()` frees the SASL connection context and all associated resources and mechanisms. The SASL connection contexts must be disposed before calling `sasl_done()`. `sasl_done()` is not responsible for releasing context resources for the SASL connection. See “libsasl Cleanup” on page 135.

When a SASL session is freed, the associated mechanisms are informed that all state can be freed. A SASL session should only be freed when the session is not to be reused. Otherwise, the SASL state can be reused by another session. Both the client and server use `sasl_dispose()` to free the SASL connection context.

libsasl Cleanup

This step releases all the resources in the SASL library and the plug-ins. The client and server call `sasl_done()` to release `libsasl()` resources and to unload all the SASL plug-ins. `sasl_done()` does not release SASL connection contexts. Note that if an application is both a SASL client and a SASL server, `sasl_done()` releases both the SASL client and SASL server resources. You cannot release the resources for just the client or the server.

Caution – Libraries should not call `sasl_done()`. Applications should exercise caution when calling `sasl_done()` to avoid interference with any libraries that might be using `libsasl`.

SASL Example

This section demonstrates a typical SASL session between a client application and server application. The example goes through these steps:

1. The client application initializes `libsasl` and sets the following global callbacks:
   - SASL_CB_GETREALM
   - SASL_CB_USER
   - SASL_CB_AUTHTITLE
   - SASL_CB_PASS
2. The server application initializes `libsasl` and sets the following global callbacks:
   - `SASL_CB_GETPATH`
   - `SASL_CB_LIST_END`

3. The client creates a SASL connection context, sets the security properties, and requests the list of available mechanisms from the server.

4. The server creates a SASL connection context, sets the security properties, gets a list of suitable SASL mechanisms, and sends the list to client.

5. The client receives the list of available mechanisms, chooses a mechanism, and sends the mechanism choice to the server together with any authentication data.

6. The client and server then exchange SASL data until the authentication and security layer negotiation is complete.

7. With the authentication complete, the client and server determine whether a security layer was negotiated. The client encodes a test message. The message is then sent to the server. The server also determines the user name of the authenticated user and the user's realm.

8. The server receives, decodes, and prints the encoded message.

9. The client calls `sasl_dispose()` to release the client's SASL connection context. The client then calls `sasl_done()` to release the `libsasl` resources.

10. The server calls `sasl_dispose()` to release the client connection context.

The dialogue between the client and the server follows. Each call to `libsasl` is displayed as the call is made. Each transfer of data is indicated by the sender and receiver. The data is displayed in encoded form preceded by the source: `C:` for the client and `S:` for server. The source code for both applications is provided in the Appendix D, “Source Code for SASL Example.”

**Client**

```
% doc-sample-client
*** Calling sasl_client_init() to initialize `libsasl` for client use ***
*** Calling sasl_client_new() to create client SASL connection context ***
*** Calling sasl_setprop() to set sasl context security properties ***
Waiting for mechanism list from server...
```

**Server**

```
% doc-sample-server digest-md5
*** Calling sasl_server_init() to initialize `libsasl` for server use ***
*** Calling sasl_server_new() to create server SASL connection context ***
*** Calling sasl_setprop() to set sasl context security properties ***
Forcing use of mechanism digest-md5
Sending list of 1 mechanism(s)
S: ZGlnZXN0LW1kNQ==
```
Client

S: ZGlzZXN0LWlkNQ==
received 10 byte message
got 'digest-md5'
Choosing best mechanism from: digest-md5
*** Calling sasl_client_start() ***
Using mechanism DIGEST-MD5
Sending initial response...
C: RElHRVNULU1ENQ==
Waiting for server reply...

Server

C: RElHRVNULU1ENQ==
got 'DIGEST-MD5'
*** Calling sasl_server_start() ***
Sending response...
S: bm9uY2U9IklcGxhRHJZNE42IgyVm5lQz15MTZ0YWxUOVcvanUrmpYMqahMM\nseb70iamOktQnNdiiHFc0OyXV0aCkxhDRoLWvbdMTlL0ngpGhlcj0i\cQ\wlj3NC0iMlxyYzOlL1heG1JZjg3yMD4L0NYzXZQ9dXHmlTgsYXvmb3JpdGhtPWIk\XNz
Waiting for client reply...

Client

S: bm9uY2U9IklcGxhRHJZNE42IgyVm5lQz15MTZ0YWxUOVcvanUrmpYMqahMM\nseb70iamOktQnNdiiHFc0OyXV0aCkxhDRoLWvbdMTlL0ngpGhlcj0i\cQ\wlj3NC0iMlxyYzOlL1heG1JZjg3yMD4L0NYzXZQ9dXHmlTgsYXvmb3JpdGhtPWIk\XNz
received 171 byte message
got 'nonce="Ibp1aDrY4N4s3z便利X2VNic9y16NalT9/wuju+rjyb7j9s",\realen="jml14142",qop="auth,auth-int,auth-conf",cipher="rc4-40,rc4-56,\rc4",maxbuf=2048,charset=utf-8,algorithm=md5-sess'
*** Calling sasl_client_step() ***
Please enter your authorization name : zzzz
Please enter your authenticator name : zzzz
Please enter your password : zz
*** Calling sasl_client_start() ***
Sending response...
C: dXNlcm5hbWU9Inp6enoiLHJlYWxtPSJqbTEwNDE0MiIsbm9uy2U9IklcGxhRHJZNE4\nVyVn5lQz15MTZ0YWxUOVcvanUrmpYMqahMM\nseb70iamOktQnNdiiHFc0OyXV0aCkxhDRoLWvbdMTlL0ngpGhlcj0i\cQ\wlj3NC0iMlxyYzOlL1heG1JZjg3yMD4L0NYzXZQ9dXHmlTgsYXvmb3JpdGhtPWIk\XNz
Waiting for server reply...

Server

C: dXNlcm5hbWU9Inp6enoiLHJlYWxtPSJqbTEwNDE0MiIsbm9uy2U9IklcGxhRHJZNE4\nVyVn5lQz15MTZ0YWxUOVcvanUrmpYMqahMM\nseb70iamOktQnNdiiHFc0OyXV0aCkxhDRoLWvbdMTlL0ngpGhlcj0i\cQ\wlj3NC0iMlxyYzOlL1heG1JZjg3yMD4L0NYzXZQ9dXHmlTgsYXvmb3JpdGhtPWIk\XNz
got 'username="zzzz",realm="jm114142",nonce="Ibp1aDrY4N4s3z便利X2VNic9y16NalT9/wuju+rjyb7j9s",\nncnonce="yyjjghlVhcDLKhhAoi1rvKCKVvYUI1C8W5rrZUZnHGvedryM",\nnc="00000001",qop=auth-conf,cipher="rc4",maxbuf=2048,digest-uri="rcmd/",\nresponse="966e978252d7f68a2zc91b2cd32a94ec"
*** Calling sasl_server_step() ***
Sending response...
S: cnXwYXV0aD0yYjEzMzRjYzU4NTE4MTEwM3OTdHd3JpYyYjOTk4MTk3OQ==
Waiting for client reply...

Client

S: cnXwYXV0aD0yYjEzMzRjYzU4NTE4MTEwM3OTdHd3JpYyYjOTk4MTk3OQ==
received 40 byte message
got 'rspauth=2b1334cc585181109c797a250b903979'
*** Calling sasl_client_step() ***
C:
Negotiation complete
*** Calling sasl_getprop() ***
Username: zzzz
SSF: 128
Waiting for encoded message...

Server

Waiting for client reply...
C: got '' *** Calling sasl_server_step() ***
Negotiation complete
*** Calling sasl_getprop() to get username, realm, ssf ***
Username: zzzz
Realm: 22c38
SSF: 128
*** Calling sasl_encode() *** sending encrypted message 'srv message 1'
S: AAAAvArjnAv0FuMBqAAxkqdmzJ6VD1oajiwABAAAAAA==

Client

S: AAAAvArjnAv0FuMBqAAxkqdmzJ6VD1oajiwABAAAAAA==
received 34 byte message
got ''
*** Calling sasl_decode() ***
received decoded message 'srv message 1'
*** Calling sasl_encode() ***
sending encrypted message 'client message 1'
C: AAAAIrdkTEMYOn9X4NXkkPC30TFvAZUnLbZAnQzn6gABAAAAAA==
*** Calling sasl_dispose() to release client SASL connection context ***
*** Calling sasl_done() to release libsasl resources ***

Server

Waiting for encrypted message...
C: AAAAIrdkTEMYOn9X4NXkkPC30TFvAZUnLbZAnQzn6gABAAAAAA==
got ''
*** Calling sasl_decode() ***
received decoded message 'client message 1'
*** Calling sasl_dispose() to release client SASL connection context ***
SASL for Service Providers

This section describes how to create plug-ins for providing mechanisms and other services to SASL applications.

Note – Due to export regulations, the Solaris SASL SPI does not support a security layer for non-Solaris client/server mechanism plug-ins. As a result, non-Solaris client/server mechanism plug-ins cannot offer integrity or privacy services. Solaris client/server mechanism plug-ins do not have this restriction.

SASL Plug-in Overview

The SASL service provider interface (SPI) enables communication between plug-ins and the libsasl library. SASL plug-ins are typically implemented as shared libraries. A single shared library can one or more SASL plug-ins of different types. Plug-ins that are in shared libraries are opened dynamically by libsasl through the dlopen(3C) function.

Plug-ins can also be statically bound to an application that calls libsasl. These kinds of plug-ins are loaded through either the sasl_client_add_plugin() function or the sasl_server_add_plugin() function, depending on whether the application is a client or server.

A SASL plug-in in the Solaris operating system has the following requirements:

- A plug-in in a shared library must be in a valid executable object file, preferably with the .so file extension.
- The plug-in must be in a location that can be verified. The SASL_CB_VERIFYFILE callback is used to verify plug-ins.
- The plug-in must contain the proper entry points.
- The version of the plug-in for the SASL client must match the version of the corresponding plug-in for the SASL server.
- The plug-in needs to be able to be initialized successfully.
- The binary type of the plug-in must match the binary type for libsasl.

SASL plug-ins fall into four categories:

- Client mechanism plug-in
- Server mechanism plug-in
- Canonicalization plug-in
- auxprop plug-in
The `sasl_client_init()` function causes SASL clients to load any available client plug-ins. The `sasl_server_init()` function causes SASL servers to load the server, canonicalization, and auxprop plug-ins. All plug-ins are unloaded when `sasl_done()` is called.

To locate plug-ins, `libsasl` uses either the `SASL_CB_GETPATH` callback function or the default path. `SASL_CB_GETPATH` returns a colon-separated list of directories to be searched for plug-ins. If the SASL consumer specifies a `SASL_CB_GETPATH` callback, then `libsasl` uses the returned path for searching. Otherwise, the SASL consumer can use the default path that corresponds to the binary type:

- 32-bit SPARC architecture: `/usr/lib/sasl`
- 32-bit x86 architecture: `/usr/lib/sasl`
- 64-bit SPARC architecture: `/usr/lib/sasl/sparcv9`
- x64 architecture: `/usr/lib/sasl/amd64`

As part of the loading process, `libsasl` calls the latest, supported version of the plug-in. The plug-in returns the version and a structure that describes the plug-in. If the version checks out, `libsasl` loads the plug-in. The current version number, `SASL_UTILS_VERSION`, is 4.

After a plug-in has been initialized, subsequent communication between the plug-in and `libsasl` takes place through structures that have to be established. Plug-ins use the `sasl_utils_t` structure to call `libsasl`. `libsasl` uses entry points in the following structures to communicate with plug-ins:

- `sasl_out_params_t`
- `sasl_client_params_t`
- `sasl_server_params_t`
- `sasl_client_plug_t`
- `sasl_server_plug_t`
- `sasl_canonuser_plug_t`
- `sasl_auxprop_plug_t`

The source code for these structures can be found in the SASL header files. The structures are described in the following section.

**Important Structures for SASL Plug-ins**

Communication between `libsasl` and plug-ins is accomplished through the following structures:

- `sasl_utils_t` - The `sasl_utils_t` structure contains a number of utility functions, along with the three contexts:
  
  This structure contains a number of utility functions that serve as a convenience for plug-in writers. Many of the functions are pointers to public interfaces in `libsasl`. Plug-ins do not need to call `libsasl` directly, unless for some reason the plug-in needs to be a SASL consumer.
libsasl creates three contexts for sasl_utils_t:

- sasl_conn_t *conn
- sasl_rand_t *rpool
- void *getopt_context

In some cases, such as loading plug-ins, the conn variable in sasl_utils_t is not actually associated with a connection. In other cases, conn is the SASL consumer’s SASL connection context. The rpool variable is used for random number generation functions. getopt_context is the context that should be used with the getopt() function.

sasl_getopt_t(3SASL), sasl_log_t(3SASL), and sasl_getcallback_t(3SASL)

- sasl_out_params_t – libsasl creates the sasl_out_params_t structure and passes the structure to mech_step() in the client or server. This structure communicates the following information to libsasl: authentication status, the authid, the authzid, maxbuf, the negotiated ssf, and information for encoding and decoding data
- sasl_client_params_t – The sasl_client_params_t structure is used by libsasl to pass the client state to a SASL client mechanism. The client mechanism’s mech_new(), mech_step(), and mech_idle() entry points are used to send this state data. The canon_user_client() entry point also requires client state to be passed along.
- sasl_server_params_t – The sasl_server_params_t structure performs a similar function to sasl_client_params_t on the server side.

Client Plug-ins

Client plug-ins are used to manage the client-side of a SASL negotiation. Client plug-ins are usually packaged with the corresponding server plug-ins. A client plug-in contains one or more client-side SASL mechanisms. Each SASL client mechanism supports authentication, and optionally integrity and confidentiality. Each mechanism provides information about that mechanism’s capabilities:

- Maximum SSF
- Maximum security flags
- Plug-in features
- Callbacks and prompt IDs for using the plug-in

Client plug-ins must export sasl_client_plug_init(). libsasl calls sasl_client_plug_init() to initialize the plug-in for the client. The plug-in returns a sasl_client_plug_t structure. The sasl_client_plug_t provides the following entry points for libsasl to call the mechanism:

- mech_new() – The client starts a connection by calling sasl_client_start(), which uses mech_new(). mech_new() performs initialization that is specific to the mechanism. If necessary, a connection context is allocated.
**Server Plug-ins**

Server plug-ins are used to manage the server-side of a SASL negotiation. Server plug-ins are usually packaged with the corresponding client plug-ins. A server plug-in contains one or more server-side SASL mechanisms. Each SASL server mechanism supports authentication, and optionally integrity and confidentiality. Each mechanism provides information about that mechanism’s capabilities:

- Maximum SSF
- Maximum security flags
- Plug-in features
- Callbacks and prompt IDs for using the plug-in

Server plug-ins must export `sasl_server_plug_init()`. `libsasl` calls `sasl_server_plug_init()` to initialize the plug-in for the server. The plug-in returns a `sasl_server_plug_t` structure. The `sasl_server_plug_t` provides the following entry points for `libsasl` to call the mechanism:

- `mech_new()` — The server starts a connection by calling `sasl_server_start()`, which uses `mech_new()`. `mech_new()` performs initialization that is specific to the mechanism. If necessary, `mech_new()` allocates a connection context.

- `mech_step()` — `mech_step()` can be called by `sasl_server_start()` and `sasl_server_step()`. `mech_step()` performs authentication on the server-side after `mech_new()` has been called. `mech_step()` returns SASL_OK if authentication is successful. SASL_CONTINUE is returned if more data is required. A SASL error code is returned if authentication fails. If an error occurs, then `seterror()` is called. If the authentication is successful, `mech_step()` must return the `sasl_out_params_t` structure with the relevant security layer information and callbacks. The `canon_user()` function is part of this structure. `canon_user()` must be called when the server receives the authentication and authorization IDs. Calling the `canon_user()` function causes `propctx` to be filled in. Any
required auxiliary property requests should be performed before the authentication is canonicalized. Authorization ID lookups are performed after the authentication is canonicalized.

The mech_step() function must fill any related sasl_out_params_t fields before SASL_OK is returned. These fields perform the following functions:

- doneflag – Indicates a complete exchange
- maxoutbuf – Indicates maximum output size for a security layer
- mechssf – Supplied SSF for the security layer
- encode() – Called by sasl_encode(), sasl_encodv(), and sasl_decode()
- decode() – Called by sasl_encode(), sasl_encodv(), and sasl_decode()
- encode_context() – Called by sasl_encode(), sasl_encodv(), and sasl_decode()
- decode_context() – Called by sasl_encode(), sasl_encodv(), and sasl_decode()
- mech_dispose() – mech_dispose() is called when the context can be safely closed.
- mech_free() – mech_free() is called when libsasl shuts down. Any remaining global state for the plug-in is freed by mech_free().
- setpass() sets a user's password. setpass() enables a mechanism to have an internal password.
- mech_avail() is called by sasl_listmech() to check if a mechanism is available for a given user. mech_avail() can create a new context and thus avoid a call to mech_new(). Use this method to create a context as long as performance is not affected.

**User Canonicalization Plug-ins**

A canonicalization plug-in provides support for alternate canonicalization of authentication and authorization names for both the client and server-side. The sasl_canonuser_plug_init() is used to load canonicalization plug-ins. A canonicalization plug-in has the following requirements:

- The canonicalized name must be copied to the output buffers.
- The same input buffer can be used as an output buffer.
- A canonicalization plug-in must function in cases where only authentication IDs or authorization IDs exist.

User canonicalization plug-ins must export a sasl_canonuser_init() function. The sasl_canonuser_init() function must return sasl_canonuser_plug_t to establish the necessary entry points. User canonicalization plug-ins must implement at least one of the canon_user_client() or canon_user_server() members of the sasl_canonuser_plug_t structure.
**Auxiliary Property (auxprop) Plug-ins**

auxprop plug-ins provide support for the lookup of auxiliary properties for both authid and authzid for a SASL server. For example, an application might want to look up the user password for an internal authentication. The `sasl_auxprop_plug_init()` function is used to initialize auxprop plug-ins and returns the `sasl_auxprop_plug_t` structure.

To implement an auxprop plug-in successfully, the `auxprop_lookup` member of the `sasl_auxprop_plug_t` structure must be implemented. The `auxprop_lookup()` function is called after canonicalization of the user name, with the canonicalized user name. The plug-in can then do any lookups that are needed for the requested auxiliary properties.

**Note** – Oracle Solaris does not currently provide auxprop plug-ins.

---

**SASL Plug-in Development Guidelines**

This section provides some additional pointers for developing SASL plug-ins.

**Error Reporting in SASL Plug-ins**

Good error reporting can help in tracking down authentication problems and in other debugging. Developers of plug-ins are encouraged to use the `sasl_seterror()` callback in the `sasl_utils_t` structure to supply detailed error information for a given connection.

**Memory Allocation in SASL Plug-ins**

The general rule for allocating memory in SASL is to free any memory that you have allocated when that memory is no longer needed. Following this rule improves performance and portability, and prevents memory leaks.

**Setting the SASL Negotiation Sequence**

A plug-in mechanism can set the order in which a client and server conduct a SASL conversation through the following flags:

- `SASL_FEAT_WANT_CLIENT_FIRST` – The client side begins the interchange.
- `SASL_FEAT_WANT_SERVER_LAST` – The server sends the final data to the client.

If neither flag is set, the mechanism plug-in sets the order internally. In this case, the mechanism must check both the client and server for data that needs to be sent. Note that the situation where the client sends first is only possible when the protocol permits an initial response.
The case in which the server sends last requires that the plug-in set *serverout when the step function returns SASL_OK. Those mechanisms that never have the server send last must set *serverout to NULL. Those mechanisms that always have the server send last need to point *serverout to the success data.
Introduction to the Oracle Solaris Cryptographic Framework

The Oracle Solaris cryptographic framework is an architecture that enables applications in the Oracle Solaris operating system to use or provide cryptographic services. All interactions with the framework are based on the RSA Security Inc. PKCS#11 Cryptographic Token Interface (Cryptoki). PKCS#11 is a product by RSA Laboratories, the research arm of RSA Security Inc. This chapter presents the following topics on the Oracle Solaris cryptographic framework:

- “Overview of the Cryptographic Framework” on page 148
- “Components of the Cryptographic Framework” on page 150
- “What Cryptography Developers Need to Know” on page 151
- “Adding Signatures to Providers” on page 273
- “Avoiding Data Cleanup Collisions in User-Level Providers” on page 152

Oracle Solaris Cryptography Terminology

An application, library, or kernel module that obtains cryptographic services is called a consumer. An application that provides cryptographic services to consumers through the framework is referred to as a provider and also as a plug-in. The software that implements a cryptographic operation is called a mechanism. A mechanism is not just the algorithm but includes the way in which the algorithm is to be applied. For example, the DES algorithm when applied to authentication is considered a separate mechanism. DES when applied to block-by-block encryption would be a different mechanism.

A token is the abstraction of a device that can perform cryptography. In addition, tokens can store information for use in cryptographic operations. A single token can support one or more mechanisms. Tokens can represent hardware, as in an accelerator board. Tokens that represent pure software are referred to as soft tokens. A token can be plugged into a slot, which continues the physical metaphor. A slot is the connecting point for applications that use cryptographic services.

In addition to specific slots for providers, the Solaris implementation provides a special slot called the metaslot. The metaslot is a component of the Solaris cryptographic framework library.
The metaslot serves as a single virtual slot with the combined capabilities of all tokens and slots that have been installed in the framework. Effectively, the metaslot enables an application to transparently connect with any available cryptographic service through a single slot. When an application requests a cryptographic service, the metaslot points to the most appropriate slot, which simplifies the process of selecting a slot. In some cases, a different slot might be required, in which case the application must perform a separate search explicitly. The metaslot is automatically enabled and can only be disabled through explicit action by the system administrator.

A session is a connection between an application that uses cryptographic services and a token. The PKCS #11 standard uses two kinds of objects: token objects and session objects. Session objects are ephemeral, that is, objects that last only for the duration of a session. Objects that persist beyond the length of a session are referred to as token objects.

The default location for token objects is $HOME/.sunw/pkcs11_softtoken. Alternatively, token objects can be stored in $SOFTTOKEN_DIR/pkcs11_softtoken. Private token objects are protected by personal identification numbers (PIN). To create or change a token object requires that the user be authenticated, unless the user is accessing a private token object.

Overview of the Cryptographic Framework

The cryptographic framework is the portion of the Oracle Solaris OS that provides cryptographic services from Oracle Corporation and from third-party suppliers. The framework provides various services:

- Message encryption and message digest
- Message authentication codes (MACs)
- Digital signing
- Application programmer interfaces (APIs) for accessing cryptographic services
- Service provider interfaces (SPIs) for providing cryptographic services
- An administration command for managing cryptographic resources

The following figure provides an overview of the cryptographic framework. The light gray shading in the figure indicates the user-level portion of the cryptographic framework. The dark gray shading represents the kernel-level portion of the framework. Private software is indicated by a background with diagonal striping.
Overview of the Cryptographic Framework

FIGURE 8–1 Overview of the Oracle Solaris Cryptographic Framework

- **Application**
- **elfsign**
- **Module verification library**
- **cryptoadm**
- **kernel crypto daemon**

**Pluggable interface**
- **libpkcs11.so**
- **pkcs11_kernel.so**
- **pkcs11_softtoken.so**

**Userland**
- **Third-party Hardware and Software pluggable tokens**
- **/dev/crypto pseudo-device driver**
- **Scheduler/load balancer**
- **Kernel programmer interface**
- **Kerberos GSS Mechanism**
- **IPsec**
- **Other kernel crypto consumers**

**Kernel**
- **Scheduler/load balancer**
- **Kernel programmer interface**
- **Third-party Hardware crypto providers**
- **Third-party Software crypto providers**

**Private components**
**User portion of cryptographic framework**
**Kernel portion of cryptographic framework**
Components of the Cryptographic Framework

The components of the cryptographic framework are described as follows.

- **libpkcs11.so** – The framework provides access through the RSA Security Inc. PKCS#11 Cryptographic Token Interface (Cryptoki). Applications need to link to the libpkcs11.so library, which implements the RSA PKCS#11 v2.11 of the standard.

- **Pluggable interface** – The pluggable interface is the service provider interface (SPI) for PKCS #11 cryptographic services that are provided by Oracle Corporation and third-party developers. Providers are user-level libraries. Providers are implemented through encryption services that are available from either hardware or software.

- **pkcs11_softtoken.so** – A private shared object that contains user-level cryptographic mechanisms that are provided by Oracle Corporation. The pkcs11_softtoken(5) library implements the RSA PKCS#11 v2.11 of the standard.

- **pkcs11_kernel.so** – The private shared object used to access kernel-level cryptographic mechanisms. pkcs11_kernel(5) implements the RSA PKCS#11 v2.11 specification. pkcs11_kernel.so offers a PKCS#11 user interface for cryptographic services that are plugged into the kernel’s service provider interface.

- **/dev/crypto pseudo device driver** – The private pseudo device driver for using kernel-level cryptographic mechanisms. This information is provided to avoid inadvertent deletion of the pseudo device driver.

- **Scheduler/load balancer** – The kernel software that is responsible for coordinating use, load balancing, and dispatching of the cryptographic service requests.

- **Kernel programmer interface** – The interface for kernel-level consumers of cryptographic services. The IPSec protocol and the kerberos GSS mechanism are typical cryptographic consumers.

  **Note** – This interface is only available through a special contract with Oracle Corporation. Send email to solaris-crypto-req ww@oracle.com for more information.

- **Oracle HW and SW cryptographic providers** – Kernel-level cryptographic services that are provided by Oracle Corporation. HW refers to hardware cryptographic services such as accelerator boards. SW refers to kernel modules that provide cryptographic services, such as an implementation of a cryptographic algorithm.

- **Kernel cryptographic framework daemon** – The private daemon that is responsible for managing system resources for cryptographic operations. The daemon is also responsible for verifying cryptographic providers.

- **Module verification library** – A private library used to verify the integrity and authenticity of all binaries that the Solaris cryptographic framework is importing.
What Cryptography Developers Need to Know

This section describes the requirements to develop the three types of applications that can plug into the Oracle Solaris cryptographic framework.

Requirements for Developers of User-Level Consumers

To develop a user-level consumer, a developer needs to keep the following items in mind:

- Include <security/cryptoki.h>.
- Make all calls through the PKCS #11 interfaces only.
- Link with libpkcs11.so.
- Libraries should not call the C_Finalize() function.

See Chapter 9, “Writing User-Level Cryptographic Applications and Providers,” for more information.

Requirements for Developers of User-Level Providers

To develop a user-level provider, a developer needs to keep the following items in mind:

- Design the provider to stand alone. Although the provider shared object need not be a full-fledged library to which applications link, all necessary symbols must exist in the provider. Assume that the provider is to be opened by dlopen(3C) in RTLD_GROUP and RTLD_NOW mode.
- Create a PKCS #11 Cryptoki implementation in a shared object. This shared object should include necessary symbols rather than depend on consumer applications.
It is highly recommended though not required to provide a \texttt{\_fini()} routine for data cleanup. This method is required to avoid collisions between \texttt{C\_Finalize()} calls when an application or shared library loads \texttt{libpkcs11} and other provider libraries concurrently. See "Avoiding Data Cleanup Collisions in User-Level Providers" on page 152.

- Apply for a certificate from Oracle Corporation. See “To Request a Certificate for Signing a Provider” on page 274.
- Use the certificate with \texttt{elfsign} to sign the binary. See "To Sign a Provider" on page 275.
- Package the shared object according to Oracle conventions. See Appendix F, "Packaging and Signing Cryptographic Providers."

### Requirements for Developers of Kernel-Level Consumers

To develop a kernel-level consumer, a developer needs to keep the following items in mind:

- Include \texttt{<sys/crypto/common.h>} and \texttt{<sys/crypto/api.h>}.  
- Make all calls through the kernel programming interface.

### Avoiding Data Cleanup Collisions in User-Level Providers

User-level libraries that plug into the cryptographic framework should supply a \texttt{\_fini()} function. The \texttt{\_fini()} function is called by the loader when the library is unloaded. The \texttt{\_fini()} function is required to ensure that all cleanup is done correctly at the right time. Libraries that use \texttt{libpkcs11} are not supposed to call \texttt{\_fini()} because \texttt{libpkcs11} is a shared library that could potentially be in use by the application.

To supply a \texttt{\_fini()} function, you need to create a \texttt{.fini} section in the program data section of a relocatable object. The \texttt{.fini} section provides a runtime termination code block. See \textit{Linker and Libraries Guide}. The following code sample demonstrates how to design a \texttt{.fini} section.

**EXAMPLE 8–1  Supplying \texttt{\_fini()} to PKCS\#11 Libraries**

```c
#pragma fini(pkcs11_fini)
static void pkcs11_fini();
/* [...] (other library code omitted) */

static void pkcs11_fini()
{
    (void) pthread_mutex_lock(&pkcs11mutex);

    /* If CRYPTO\_\_\_ initialized, do not clean up */
    if (!initialiazed) {
        (void) pthread_mutex_unlock(&pkcs11mutex);
    }
```

---

Developer's Guide to Oracle Solaris 10 Security • September 2013
EXAMPLE 8-1  Supplying _fini() to PKCS #11 Libraries  (Continued)

```c
return;
}

(void) finalize_routine(NULL_PTR);
(void) pthread_mutex_unlock(&pkcs11mutex);
```
This chapter explains how to develop user-level applications and providers that use the PKCS #11 functions for cryptography. The following topics are covered:

- “PKCS #11 Function List” on page 156
- “Functions for Using PKCS #11” on page 156
- “Message Digest Example” on page 163
- “Symmetric Encryption Example” on page 165
- “Sign and Verify Example” on page 170
- “Random Byte Generation Example” on page 176

For more information about the cryptographic framework, refer to Chapter 8, "Introduction to the Oracle Solaris Cryptographic Framework."

Overview of the Cryptoki Library

User-level applications in the Oracle Solaris cryptographic framework access PKCS #11 functions through the cryptoki library, which is provided in the `libpkcs11.so` module. The `pkcs11_softtoken.so` module is a PKCS #11 Soft Token implementation that is provided by Oracle Corporation to supply cryptographic mechanisms. The soft token plug-in is the default source of mechanisms. Cryptographic mechanisms can also be supplied through third-party plug-ins.

This section lists the PKCS #11 functions and return values that are supported by the soft token. Return codes vary depending on the providers that are plugged into the framework. The section also describes some common functions. For a complete description of all the elements in the cryptoki library, refer to `libpkcs11(3LIB)` or to PKCS #11: Cryptographic Token Interface Standard on the RSA Laboratories web site.
PKCS #11 Function List

The following list shows the categories of PKCS #11 functions that are supported by pkcs11_softtoken.so in the Solaris cryptographic framework with the associated functions:

- **General purpose** – C_Initialize(), C_Finalize(), C_GetInfo(), C_GetFunctionList()
- **Session management** – C_OpenSession(), C_CloseSession(), C_GetSessionInfo(), C_CloseAllSessions(), C_Login(), C_Logout()
- **Slot and token management** – C_GetSlotList(), C_GetSlotInfo(), C_GetMechanismList(), C_GetMechanismInfo(), C_SetPIN()
- **Encryption and decryption** – C_EncryptInit(), C_Encrypt(), C_EncryptUpdate(), C_EncryptFinal(), C_DecryptInit(), C_Decrypt(), C_DecryptUpdate(), C_DecryptFinal()
- **Message digesting** – C_DigestInit(), C_Digest(), C_DigestKey(), C_DigestUpdate(), C_DigestFinal()
- **Signing and applying MAC** – C_Sign(), C_SignInit(), C_SignUpdate(), C_SignFinal(), C_SignRecoverInit(), C_SignRecover()
- **Signature verification** – C_Verify(), C_VerifyInit(), C_VerifyUpdate(), C_VerifyFinal(), C_VerifyRecoverInit(), C_VerifyRecover()
- **Dual-purpose cryptographic functions** – C_DigestEncryptUpdate(), C_DecryptDigestUpdate(), C_SignEncryptUpdate(), C_DecryptVerifyUpdate()
- **Random number generation** – C_SeedRandom(), C_GenerateRandom()
- **Object management** – C_CreateObject(), C_DestroyObject(), C_CopyObject(), C_FindObjects(), C_FindObjectsInit(), C_FindObjectsFinal(), C_GetAttributeValue(), C_SetAttributeValue()
- **Key management** – C_GenerateKey(), C_GenerateKeyPair(), C_DeriveKey()

Functions for Using PKCS #11

This section provides descriptions of the following functions for using PKCS #11:

- “PKCS #11 Functions: C_Initialize()” on page 157
- “PKCS #11 Functions: C_GetInfo()” on page 157
- “PKCS #11 Functions: C_GetSlotList()” on page 158
- “PKCS #11 Functions: C_GetTokenInfo()” on page 158
- “PKCS #11 Functions: C_OpenSession()” on page 159
- “PKCS #11 Functions: C_GetMechanismList()” on page 160
Note – All the PKCS #11 functions are available from libpkcs11.so library. You do not have to use the `C_GetFunctionList()` function to get the list of functions available.

**PKCS #11 Functions: C_Initialize()**

C_Initialize() initializes the PKCS #11 library. C_Initialize() uses the following syntax:

```c
C_Initialize(CK_VOID_PTR pInitArgs);
```

`pInitArgs` is either the null value `NULL_PTR` or else a pointer to a `CK_C_INITIALIZE_ARGS` structure. With `NULL_PTR`, the library uses the Solaris mutexes as locking primitives to arbitrate the access to internal shared structures between multiple threads. Note that the Solaris cryptographic framework does not accept mutexes. Because this implementation of the cryptoki library handles multithreading safely and efficiently, using `NULL_PTR` is recommended. An application can also use `pInitArgs` to set flags such as `CKF_LIBRARY_CANT_CREATE_OS_THREADS`. C_Finalize() signals that the application is through with the PKCS #11 library.

Note – C_Finalize() should never be called by libraries. By convention, applications are responsible for calling C_Finalize() to close out a session.

In addition to `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, and `CKR_OK`, C_Initialize() uses the following return values:

- `CKR_ARGUMENTS_BAD`
- `CKR_CANT_LOCK`
- `CKR_CRYPTOKI_ALREADY_INITIALIZED` – This error is nonfatal.

**PKCS #11 Functions: C_GetInfo()**

C_GetInfo() uses manufacturer and version information about the cryptoki library. C_GetInfo() uses the following syntax:

```c
C_GetInfo(CK_INFO_PTR pInfo);
```

C_GetInfo() returns the following values:

- cryptokiVersion = 2, 11
- manufacturerID = Sun Microsystems, Inc.

In addition to `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, and `CKR_OK`, C_GetInfo() gets the following return values:

- `CKR_ARGUMENTS_BAD`
- `CKR_CRYPTOKI_NOT_INITIALIZED`
**PKCS#11 Functions: C_GetSlotList()**

C_GetSlotList() uses a list of available slots. If no additional cryptographic providers have been installed other than pkcs11_softtoken.so, then C_GetSlotList() returns the default slot only. C_GetSlotList() uses the following syntax:

```c
C_GetSlotList(CK_BBOOL tokenPresent, CK_SLOT_ID_PTR pSlotList, CK_ULONG_PTR pulCount);
```

When set to TRUE, `tokenPresent` limits the search to those slots whose tokens are present.

When `pSlotList` is set to NULL_PTR, C_GetSlotList() returns the number of slots only. `pulCount` is a pointer to the location to receive the slot count.

When `pSlotList` points to the buffer to receive the slots, `pulCount` is set to the maximum expected number of CK_SLOT_ID elements. On return, `pulCount` is set to the actual number of CK_SLOT_ID elements.

Typically, PKCS#11 applications call C_GetSlotList() twice. The first time, C_GetSlotList() is called to get the number of slots for memory allocation. The second time, C_GetSlotList() is called to retrieve the slots.

---

**Note** – The order of the slots is not guaranteed. The order of the slots can vary with each load of the PKCS#11 library.

In addition to CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, and CKR_OK, C_GetSlotList() gets the following return values:

- CKR_ARGUMENTS_BAD
- CKR_BUFFER_TOO_SMALL
- CKR_CRYPTOKI_NOT_INITIALIZED

**PKCS#11 Functions: C_GetTokenInfo()**

C_GetTokenInfo() gets information about a specific token. C_GetTokenInfo() uses the following syntax:

```c
C_GetTokenInfo(CK_SLOT_ID slotID, CK_TOKEN_INFO_PTR pInfo);
```

`slotID` identifies the slot for the token. `slotID` has to be a valid ID that was returned by C_GetSlotList(). `pInfo` is a pointer to the location to receive the token information.

If pkcs11_softtoken.so is the only installed provider, then C_GetTokenInfo() returns the following fields and values:

- **label** – Sun Software PKCS#11 softtoken.
- **flags** – CKF_DUAL_CRYPTO_OPERATIONS, CKF_TOKEN_INITIALIZED, CKF_RNG, CKF_USER_PIN_INITIALIZED, and CKF_LOGIN_REQUIRED, which are set to 1.
ulMaxSessionCount – Set to CK_EFFECTIVELY_INFINITE.
ulMaxRwSessionCount - Set to CK_EFFECTIVELY_INFINITE.
ulMaxPinLen – Set to 256.
ulMinPinLen – Set to 1.
ulTotalPublicMemory set to CK_UNAVAILABLE_INFORMATION
ulFreePublicMemory set to CK_UNAVAILABLE_INFORMATION
ulTotalPrivateMemory set to CK_UNAVAILABLE_INFORMATION
ulFreePrivateMemory set to CK_UNAVAILABLE_INFORMATION

In addition to CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, and CKR_OK, C_GetSlotlist() gets the following return values:

- CKR_ARGUMENTS_BAD
- CKR_BUFFER_TOO_SMALL
- CKR_CRYPTOKI_NOT_INITIALIZED
- CKR_SLOT_ID_INVALID

The following return values are relevant for plug-ins with hardware tokens:

- CKR_DEVICE_ERROR
- CKR_DEVICE_MEMORY
- CKR_DEVICE_REMOVED
- CKR_TOKEN_NOT_PRESENT
- CKR_TOKEN_NOT_RECOGNIZED

PKCS #11 Functions: C_OpenSession()

C_OpenSession() enables an application to start a cryptographic session with a specific token in a specific slot. C_OpenSession() uses the following syntax:

C_OpenSession(CK_SLOT_ID slotID, CK_FLAGS flags, CK_VOID_PTR pApplication, CK_NOTIFY Notify, CK_SESSION_HANDLE_PTR phSession);

slotID identifies the slot. flags indicates whether the session is read-write or read-only. pApplication is a pointer that is defined by the application for use in callbacks. Notify holds the address of an optional callback function. phSession is a pointer to the location of the session handle.

In addition to CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, and CKR_OK, C_OpenSession() gets the following return values:

- CKR_ARGUMENTS_BAD
- CKR_CRYPTOKI_NOT_INITIALIZED
- CKR_SLOT_ID_INVALID
- CKR_TOKEN_WRITE_PROTECTED – Occurs with write-protected tokens.
The following return values are relevant for plug-ins with hardware tokens:

- CKR_DEVICE_ERROR
- CKR_DEVICE_MEMORY
- CKR_DEVICE_REMOVED
- CKR_SESSION_COUNT
- CKR_SESSION_PARALLEL_NOT_SUPPORTED
- CKR_SESSION_READ_WRITE_SO_EXISTS
- CKR_TOKEN_NOT_PRESENT
- CKR_TOKEN_NOT_RECOGNIZED

**PKCS #11 Functions: C_GetMechanismList()**

C_GetMechanismList() gets a list of mechanism types that are supported by the specified token. C_GetMechanismList() uses the following syntax:

```c
C_GetMechanismList(CK_SLOT_ID slotID, CK_MECHANISM_TYPE_PTR pMechanismList,
                    CK_ULONG_PTR pulCount);
```

slotID identifies the slot for the token. pulCount is a pointer to the location to receive the number of mechanisms. When pMechanismList is set to NULL_PTR, the number of mechanisms is returned in *pulCount. Otherwise, *pulCount must be set to the size of the list and pMechanismList points to the buffer to hold the list.

When PKCS #11 Soft Token is plugged in, C_GetMechanismList() returns the following list of supported mechanisms:

- CKM_AES_CBC
- CKM_AES_CBC_PAD
- CKM_AES_ECB
- CKM_AES_KEY_GEN
- CKM_DES_CBC
- CKM_DES_CBC_PAD
- CKM_DES_ECB
- CKM_DES_KEY_GEN
- CKM_DES_MAC
- CKM_DES_MAC_GENERAL
- CKM_DES3_CBC
- CKM_DES3_CBC_PAD
- CKM_DES3_ECB
- CKM_DES3_KEY_GEN
- CKM_DH_PKCS_DERIVE
- CKM_DH_PKCS_KEY_PAIR_GEN
- CKM_DSA
- CKM_DSA_KEY_PAIR_GEN
- CKM_DSA_SHA_1
- CKM_MD5
In addition to CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, and CKR_OK, C_GetSlotlist() uses the following return values:

- CKR_ARGUMENTS_BAD
- CKR_BUFFER_TOO_SMALL
- CKR_CRYPTOKI_NOT_INITIALIZED
- CKR_SLOT_ID_INVALID

The following return values are relevant for plug-ins with hardware tokens:

- CKR_DEVICE_ERROR
- CKR_DEVICE_MEMORY
- CKR_DEVICE_REMOVED
- CKR_TOKEN_NOT_PRESENT
- CKR_TOKEN_NOT_RECOGNIZED
Extended PKCS #11 Functions

In addition to the standard PKCS #11 functions, two convenience functions are supplied with the Oracle Solaris cryptographic framework:

- “Extended PKCS #11 Functions: SUNW_C_GetMechSession()” on page 162
- “Extended PKCS #11 Functions: SUNW_C_KeyToObject” on page 162

Extended PKCS #11 Functions: SUNW_C_GetMechSession()

SUNW_C_GetMechSession() is a convenience function that initializes the Solaris cryptographic framework. The function then starts a session with the specified mechanism.

SUNW_C_GetMechSession() uses the following syntax:

```c
SUNW_C_GetMechSession(CK_MECHANISM_TYPE mech, CK_SESSION_HANDLE_PTR hSession)
```

The `mech` parameter is used to specify the mechanism to be used. `hSession` is a pointer to the session location.

Internally, SUNW_C_GetMechSession() calls C_Initialize() to initialize the cryptoki library. SUNW_C_GetMechSession() next calls C_GetSlotList() and C_GetMechanismInfo() to search through the available slots for a token with the specified mechanism. When the mechanism is found, SUNW_C_GetMechSession() calls C_OpenSession() to open a session.

The SUNW_C_GetMechSession() only needs to be called once. However, calling SUNW_C_GetMechSession() multiple times does not cause any problems.

Extended PKCS #11 Functions: SUNW_C_KeyToObject

SUNW_C_KeyToObject() creates a secret key object. The calling program must specify the mechanism to be used and raw key data. Internally, SUNW_C_KeyToObject() determines the type of key for the specified mechanism. A generic key object is created through C_CreateObject(). SUNW_C_KeyToObject() next calls C_GetSessionInfo() and C_GetMechanismInfo() to get the slot and mechanism. C_SetAttributeValue() then sets the attribute flag for the key object according to the type of mechanism.

User-Level Cryptographic Application Examples

This section includes the following examples:

- “Message Digest Example” on page 163
- “Symmetric Encryption Example” on page 165
- “Sign and Verify Example” on page 170
- “Random Byte Generation Example” on page 176
**Message Digest Example**

This example uses PKCS #11 functions to create a digest from an input file. The example performs the following steps:

1. Specifies the digest mechanism.
   - In this example, the CKM_MD5 digest mechanism is used.
2. Finds a slot that is capable of the specified digest algorithm.
   - This example uses the Oracle Solaris convenience function SUNW_C_GetMechSession(). SUNW_C_GetMechSession() opens the cryptoki library, which holds all the PKCS #11 functions that are used in the Oracle Solaris cryptographic framework. SUNW_C_GetMechSession() then finds the slot with the desired mechanism. The session is then started. Effectively, this convenience function replaces the C_Initialize() call, the C_OpenSession() call, and any code needed to find a slot that supports the specified mechanism.
3. Obtains cryptoki information.
   - This part is not actually needed to create the message digest, but is included to demonstrate use of the C_GetInfo() function. This example gets the manufacturer ID. The other information options retrieve version and library data.
4. Conducts a digest operation with the slot.
   - The message digest is created in this task through these steps:
     - Opening the input file.
     - Initializing the digest operation by calling C_DigestInit().
     - Processing the data a piece at a time with C_DigestUpdate().
     - Ending the digest process by using C_DigestFinal() to get the complete digest.
5. Ends the session.
   - The program uses C_CloseSession() to close the session and C_Finalize() to close the library.

The source code for the message digest example is shown in the following example.

**EXAMPLE 9–1  Creating a Message Digest Using PKCS #11 Functions**

```c
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <security/cryptoki.h>
#include <security/pkcs11.h>

#define BUFFERSIZ 8192
#define MAXDIGEST 64

/* Calculate the digest of a user supplied file. */
void
```
main(int argc, char **argv)
{
    CK_BYTE digest[MAXDIGEST];
    CK_INFO info;
    CK_MECHANISM mechanism;
    CK_SESSION_HANDLE hSession;
    CK_SESSION_INFO Info;
    CK_ULONG ulDatalen = BUFFERSIZ;
    CK_ULONG ulDigestLen = MAXDIGEST;
    CK_RV rv;
    CK_SLOT_ID SlotID;

    int i, bytes_read = 0;
    char inbuf[BUFFERSIZ];
    FILE *fs;
    int error = 0;

    /* Specify the CKM_MDS digest mechanism as the target */
    mechanism.mechanism = CKM_MD5;
    mechanism.pParameter = NULL_PTR;
    mechanism.ulParameterLen = 0;

    /* Use SUNW convenience function to initialize the cryptoki *
    * library, and open a session with a slot that supports
    * the mechanism we plan on using. */
    rv = SUNW_C_GetMechSession(mechanism.mechanism, &hSession);
    if (rv != CKR_OK) {
        fprintf(stderr, "SUNW_C_GetMechSession: rv = 0x%.8X\n", rv);
        exit(1);
    }

    /* Get cryptoki information, the manufacturer ID */
    rv = C_GetInfo(&info);
    if (rv != CKR_OK) {
        fprintf(stderr, "WARNING: C_GetInfo: rv = 0x%.8X\n", rv);
    }
    fprintf(stdout, "Manufacturer ID = %s\n", info.manufacturerID);

    /* Open the input file */
    if ((fs = fopen(argv[1], "r")) == NULL) {
        perror("fopen");
        fprintf(stderr, "\n\nusage: %s filename\n", argv[0]);
        error = 1;
        goto exit_session;
    }

    /* Initialize the digest session */
    if ((rv = C_DigestInit(hSession, &mechanism)) != CKR_OK) {
        fprintf(stderr, "C_DigestInit: rv = 0x%.8X\n", rv);
        error = 1;
        goto exit_digest;
    }

    /* Read in the data and create digest of this portion */
    while (!feof(fs) && (ulDatalen = fread(inbuf, 1, BUFFERSIZ, fs)) > 0) {
        if ((rv = C_DigestUpdate(hSession, (CK_BYTE_PTR)inbuf,
            User-Level Cryptographic Application Examples

Developer's Guide to Oracle Solaris 10 Security  •  September 2013
EXAMPLE 9–1  Creating a Message Digest Using PKCS #11 Functions  (Continued)

```c
ulDatalen) != CKR_OK) {
    fprintf(stderr, "C_DigestUpdate: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_digest;
} bytes_read += ulDatalen;
}
fprintf(stdout, "%d bytes read and digested!!!\n\n", bytes_read);
/* Get complete digest */
ulDigestLen = sizeof (digest);
if ((rv = C_DigestFinal(hSession, (CK_BYTE_PTR)digest,
    &ulDigestLen)) != CKR_OK) {
    fprintf(stderr, "C_DigestFinal: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_digest;
}
/* Print the results */
fprintf(stdout, "The value of the digest is: ");
for (i = 0; i < ulDigestLen; i++) {
    fprintf(stdout, "%.2x", digest[i]);
}
fprintf(stdout, "\nDone!!!\n");
exit_digest:
    fclose(fs);
exit_session:
    (void) C_CloseSession(hSession);
exit_program:
    (void) C_Finalize(NULL_PTR);
exit(error);
```

Symmetric Encryption Example

Example 9–2 creates a key object for encryption with the DES algorithm in the CBC mode. This source code performs the following steps:

1. Declares key materials.
   Defines DES and initialization vector. The initialization vector is declared statically for demonstration purposes only. Initialization vectors should always be defined dynamically and never reused.

2. Defines a key object.
   For this task, you have to set up a template for the key.

3. Finds a slot that is capable of the specified encryption mechanism.
This example uses the Oracle Solaris convenience function \texttt{SUNW\_C\_GetMechSession()}. \texttt{SUNW\_C\_GetMechSession()} opens the \texttt{cryptoki} library, which holds all the PKCS #11 functions that are used in the Oracle Solaris cryptographic framework. \texttt{SUNW\_C\_GetMechSession()} then finds the slot with the desired mechanism. The session is then started. Effectively, this convenience function replaces the \texttt{C\_Initialize()} call, the \texttt{C\_OpenSession()} call, and any code needed to find a slot that supports the specified mechanism.

4. Conducts an encryption operation in the slot.

The encryption is performed in this task through these steps:

a. Opening the input file.

b. Creating an object handle for the key.

c. Setting the encryption mechanism to \texttt{CKM\_DES\_CBC\_PAD} by using the mechanism structure.

d. Initializing the encryption operation by calling \texttt{C\_EncryptInit()}.

e. Processing the data piece at a time with \texttt{C\_EncryptUpdate()}.

f. Ending the encryption process by using \texttt{C\_EncryptFinal()} to get the last portion of the encrypted data.

5. Conducts a decryption operation in the slot.

The decryption is performed in this task through these steps. The decryption is provided for testing purposes only.

a. Initializes the decryption operation by calling \texttt{C\_DecryptInit()}.

b. Processes the entire string with \texttt{C\_Decrypt()}.

6. Ends the session.

The program uses \texttt{C\_CloseSession()} to close the session and \texttt{C\_Finalize()} to close the library.

The source code for the symmetric encryption example is shown in the following example.

\textbf{EXAMPLE 9-2}  Creating an Encryption Key Object Using PKCS #11 Functions

\begin{verbatim}
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <security/cryptoki.h>
#include <security/pkcs11.h>

#define BUFFERSIZ 8192

/* Declare values for the key materials. DO NOT declare initialization vectors statically like this in real life!! */
uchar_t des_key[] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef};
uchar_t des_cbc_iv[] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab, 0xcd, 0xef};
\end{verbatim}
/* Key template related definitions. */
static CK_BBOOL truevalue = TRUE;
static CK_BBOOL falsevalue = FALSE;
static CK_OBJECT_CLASS class = CKO_SECRET_KEY;
static CK_KEY_TYPE keyType = CKK_DES;

/* Example encrypts and decrypts a file provided by the user. */
void main(int argc, char **argv)
{
    CK_RV rv;
    CK_MECHANISM mechanism;
    CK_OBJECT_HANDLE hKey;
    CK_SESSION_HANDLE hSession;
    CK_ULONG ciphertext_len = 64, lastpart_len = 64;
    long ciphertext_space = BUFFERSIZ;
    CK_ULONG decrypttext_len;
    CK_ULONG total_encrypted = 0;
    CK_ULONG ulDatalen = BUFFERSIZ;
    int i, bytes_read = 0;
    int error = 0;
    char inbuf[BUFFERSIZ];
    FILE *fs;
    uchar_t ciphertext[BUFFERSIZ], *pciphertext, decrypttext[BUFFERSIZ];

    /* Set the key object */
    CK_ATTRIBUTE template[] = {
        {CKA_CLASS, &class, sizeof (class) },
        {CKA_KEY_TYPE, &keyType, sizeof (keyType) },
        {CKA_TOKEN, &falsevalue, sizeof (falsevalue) },
        {CKA_ENCRYPT, &truevalue, sizeof (truevalue) },
        {CKA_VALUE, &des_key, sizeof (des_key) }
    };

    /* Set the encryption mechanism to CKM DES_CBC_CBC_PAD */
    mechanism.mechanism = CKM_DES_CBC_CBC_PAD;
    mechanism.pParameter = des_cbc_iv;
    mechanism.ulParameterLen = 8;

    /* Use SUNW convenience function to initialize the cryptoki */
    /* library, and open a session with a slot that supports */
    /* the mechanism we plan on using. */
    rv = SUNW_C_GetMechSession(mechanism.mechanism, &hSession);

    if (rv != CKR_OK) {
        fprintf(stderr, "SUNW_C_GetMechSession: rv = 0x%.8X\n", rv);
        exit(1);
    }

    /* Open the input file */
    if ((fs = fopen(argv[1], "r")) == NULL) {
        perror("fopen");
        fprintf(stderr, "\n\nusage: %s filename\n", argv[0]);
        error = 1;
    }
EXAMPLE 9–2 Creating an Encryption Key Object Using PKCS #11 Functions (Continued)

goto exit_session;
}

/* Create an object handle for the key */
rv = C_CreateObject(hSession, template,
 sizeof (template) / sizeof (CK_ATTRIBUTE),
 &hKey);
if (rv != CKR_OK) {
 fprintf(stderr, "C_CreateObject: rv = 0x%.8X\n", rv);
 error = 1;
 goto exit_session;
}

/* Initialize the encryption operation in the session */
rv = C_EncryptInit(hSession, &mechanism, hKey);
if (rv != CKR_OK) {
 fprintf(stderr, "C_EncryptInit: rv = 0x%.8X\n", rv);
 error = 1;
 goto exit_session;
}

/* Read in the data and encrypt this portion */
pcipher_text = &cipher_text[0];
while (!feof(fs) && (cipher_text_space > 0) &&
 (ulDatalen = fread(inbuf, 1, cipher_text_space, fs)) > 0) {
 cipher_text_len = cipher_text_space;

 /* C_EncryptUpdate is only being sent one byte at a
  * time, so we are not checking for CKR_BUFFER_TOO_SMALL.
  * Also, we are checking to make sure we do not go
  * over the alloted buffer size. A more robust program
  * could incorporate realloc to enlarge the buffer
  * dynamically. */
  rv = C_EncryptUpdate(hSession, (CK_BYTE_PTR)inbuf, ulDatalen,
   &cipher_text, &cipher_text_len);
  if (rv != CKR_OK) {
   fprintf(stderr, "C_EncryptUpdate: rv = 0x%.8X\n", rv);
   error = 1;
   goto exit_encrypt;
  }
  pcipher_text += cipher_text_len;
  total_encrypted += cipher_text_len;
  cipher_text_space -= cipher_text_len;
  bytes_read += ulDatalen;
}
if (!feof(fs) || (cipher_text_space < 0)) {
 fprintf(stderr, "Insufficient space for encrypting the file\n");
 error = 1;
 goto exit_encrypt;
}

/* Get the last portion of the encrypted data */
EXAMPLE 9–2  Creating an Encryption Key Object Using PKCS #11 Functions  (Continued)

```c
lastpart_len = ciphertext_space;
rv = C_EncryptFinal(hSession, pciphertext, &lastpart_len);
if (rv != CKR_OK) {
    fprintf(stderr, "C_EncryptFinal: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_encrypt;
}
total_encrypted += lastpart_len;

fprintf(stdout, "%d bytes read and encrypted. Size of the "
    "ciphertext: %d!\n\n", bytes_read, total_encrypted);
/* Print the encryption results */
fprintf(stdout, "The value of the encryption is:\n");
for (i = 0; i < ciphertext_len; i++) {
    if (ciphertext[i] < 16)
        fprintf(stdout, "0%x", ciphertext[i]);
    else
        fprintf(stdout, "%2x", ciphertext[i]);
}
/* Initialize the decryption operation in the session */
rv = C_DecryptInit(hSession, &mechanism, hKey);
/* Decrypt the entire ciphertext string */
decrypttext_len = sizeof (decrypttext);
rv = C_Decrypt(hSession, (CK_BYTE_PTR)ciphertext, total_encrypted,
    decrypttext, &decrypttext_len);
if (rv != CKR_OK) {
    fprintf(stderr, "C_Decrypt: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_encrypt;
}
fprintf(stdout, "%d bytes decrypted!!!\n\n", decrypttext_len);
/* Print the decryption results */
printf(stdout, "The value of the decryption is:\n%s", decrypttext);
printf(stdout, "\nDone!!!\n");
exit_encrypt:
    fclose(fs);
exit_session:
    (void) C_CloseSession(hSession);
exit_program:
    (void) C_Finalize(NULL_PTR);
    exit(error);
```
Sign and Verify Example

The example in this section generates an RSA key pair. The key pair is used to sign and verify a simple string. The example goes through the following steps:

1. Defines a key object.
2. Sets the public key template.
3. Sets the private key template.
4. Creates a sample message.
5. Specifies the genmech mechanism, which generates the key pair.
6. Specifies the smech mechanism, which signs the key pair.
7. Initializes the cryptoki library.
8. Finds a slot with mechanisms for signing, verifying, and key pair generation. The task uses a function that is called getMySlot(), which performs the following steps:
   a. Calling the function C_GetSlotList() to get a list of the available slots.
      C_GetSlotList() is called twice, as the PKCS #11 convention suggests.
      C_GetSlotList() is called the first time to get the number of slots for memory allocation. C_GetSlotList() is called the second time to retrieve the slots.
   b. Finding a slot that can supply the desired mechanisms.
      For each slot, the function calls GetMechanismInfo() to find mechanisms for signing and for key pair generation. If the mechanisms are not supported by the slot, GetMechanismInfo() returns an error. If GetMechanismInfo() returns successfully, then the mechanism flags are checked to make sure the mechanisms can perform the needed operations.
9. Opens the session by calling C_OpenSession().
10. Generates the key pair by using C_GenerateKeyPair().
11. Displays the public key with C_GetAttributeValue() – For demonstration purposes only.
12. Signing is started with C_SignInit() and completed with C_Sign().
13. Verification is started with C_VerifyInit() and completed with C_Verify().
14. Closes the session.
   The program uses C_CloseSession() to close the session and C_Finalize() to close the library.

The source code for the sign-and-verify example follows.

EXAMPLE 9–3  Signing and Verifying Text Using PKCS #11 Functions
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <cryptoki.h>
#include <sys/types.h>
#include <security/cryptoki.h>
#include <security/pkcs11.h>

#define BUFFERSIZ 8192

/* Define key template */
static CK_BBOOL truevalue = TRUE;
static CK_BBOOL falsevalue = FALSE;
static CK_ULONG modulusbits = 1024;
static CK_BYTE public_exponent[] = {3};

boolean_t GetMySlot(CK_MECHANISM_TYPE sv_mech, CK_MECHANISM_TYPE kpgen_mech,
                     CK_SLOT_ID_PTR pslot);

/* Example signs and verifies a simple string, using a public/private
 * key pair. */
void main(int argc, char **argv)
{
    CK_RV rv;
    CK_MECHANISM genmech, smech;
    CK_SESSION_HANDLE hSession;
    CK_SESSION_INFO sessInfo;
    CK_SLOT_ID slotID;
    int error, i = 0;
    CK_OBJECT_HANDLE privatekey, publickey;

    /* Set public key. */
    CK_ATTRIBUTE publickey_template[] = {
        {CKA_VERIFY, &truevalue, sizeof (truevalue)},
        {CKA_MODULUS_BITS, &modulusbits, sizeof (modulusbits)},
        {CKA_PUBLIC_EXPONENT, &public_exponent,
            sizeof (public_exponent)}
    };

    /* Set private key. */
    CK_ATTRIBUTE privatekey_template[] = {
        {CKA_SIGN, &truevalue, sizeof (truevalue)},
        {CKA_TOKEN, &falsevalue, sizeof (falsevalue)},
        {CKA_SENSITIVE, &truevalue, sizeof (truevalue)},
        {CKA_EXTRACTABLE, &truevalue, sizeof (truevalue)}
    };

    /* Create sample message. */
    CK_ATTRIBUTE getattributes[] = {
        {CKA_MODULUS_BITS, NULL_PTR, 0},
        {CKA_MODULUS, NULL_PTR, 0},
        {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
    };

    CK_ULONG messagelen, slen, template_size;
    boolean_t found_slot = B_FALSE;
    uchar_t *message = (uchar_t *)
        "Simple message for signing & verifying."
            "Hello, world!"
        "Simple message for signing & verifying."
    ;
EXAMPLE 9–3  Signing and Verifying Text Using PKCS #11 Functions  (Continued)

uchar_t *modulus, *pub_exponent;
char sign[BUFFERSIZ];
slen = BUFFERSIZ;

messagelen = strlen((char *)message);

/* Set up mechanism for generating key pair */
genmech.mechanism = CKM_RSA_PKCS_KEY_PAIR_GEN;
genmech.pParameter = NULL_PTR;
genmech.ulParameterLen = 0;

/* Set up the signing mechanism */
smech.mechanism = CKM_RSA_PKCS;
smech.pParameter = NULL_PTR;
smech.ulParameterLen = 0;

/* Initialize the CRYPTOKI library */
rv = C_Initialize(NULL_PTR);

if (rv != CKR_OK) {
    fprintf(stderr, "C_Initialize: Error = 0x%.8X\n", rv);
    exit(1);
}

found_slot = GetMySlot(smech.mechanism, genmech.mechanism, &slotID);

if (!found_slot) {
    fprintf(stderr, "No usable slot was found.\n");
    goto exit_program;
}

fprintf(stdout, "selected slot: %d\n", slotID);

/* Open a session on the slot found */
rv = C_OpenSession(slotID, CKF_SERIAL_SESSION, NULL_PTR, NULL_PTR,
&hSession);

if (rv != CKR_OK) {
    fprintf(stderr, "C_OpenSession: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_program;
}

fprintf(stdout, "Generating keypair....\n");

/* Generate Key pair for signing/verifying */
rv = C_GenerateKeyPair(hSession, &genmech, publickey_template,
(sizeof (publickey_template) / sizeof (CK_ATTRIBUTE)),
privatekey_template,
(sizeof (privatekey_template) / sizeof (CK_ATTRIBUTE)),
&publickey, &privatekey);

if (rv != CKR_OK) {
    fprintf(stderr, "C_GenerateKeyPair: rv = 0x%.8X\n", rv);
    error = 1;
    goto exit_session;

/* Display the public key. */
template_size = sizeof (getattributes) / sizeof (CK_ATTRIBUTE);
rv = C_GetAttributeValue(hSession, publickey, getattributes,
    template_size);

if (rv != CKR_OK) {
    /* not fatal, we can still sign/verify if this failed */
    fprintf(stderr, "C_GetAttributeValue: rv = 0x%.8X\n", rv);
    error = 1;
} else {
    /* Allocate memory to hold the data we want */
    for (i = 0; i < template_size; i++) {
        getattributes[i].pValue = malloc (getattributes[i].ulValueLen *
            sizeof(CK_VOID_PTR));
        if (getattributes[i].pValue == NULL) {
            int j;
            for (j = 0; j < i; j++)
                free(getattributes[j].pValue);
            goto sign_cont;
        }
    }
    /* Call again to get actual attributes */
    rv = C_GetAttributeValue(hSession, publickey, getattributes,
        template_size);
    if (rv != CKR_OK) {
        /* not fatal, we can still sign/verify if failed */
        fprintf(stderr, "C_GetAttributeValue: rv = 0x%.8X\n", rv);
        error = 1;
    } else {
        /* Display public key values */
        fprintf(stdout, "Public Key data:\n\tModulus bits: \n%ld\n", *(CKULONG_PTR)(getattributes[0].pValue));
        fprintf(stdout, "\tModulus: \n%\n", modulus = (uchar_t *)getattributes[1].pValue;
        for (i = 0; i < getattributes[1].ulValueLen; i++)
            fprintf(stdout, "%.2x", modulus[i]);
        fprintf(stdout, "\n\tPublic Exponent: \n%\n", pub_exponent = (uchar_t *)getattributes[2].pValue;
        for (i = 0; i < getattributes[2].ulValueLen; i++)
            fprintf(stdout, "%.2x", pub_exponent[i]);
        fprintf(stdout, "\n\n");
    }
}
EXAMPLE 9-3  Signing and Verifying Text Using PKCS#11 Functions  (Continued)

```c
sign_cont:
    rv = C_SignInit(hSession, &smech, privatekey);

    if (rv != CKR_OK) {
        fprintf(stderr, "C_SignInit: rv = 0x%.8X\n", rv);
        error = 1;
        goto exit_session;
    }

    rv = C_Sign(hSession, (CK_BYTE_PTR)message, messagelen,
                 (CK_BYTE_PTR)sign, &slen);

    if (rv != CKR_OK) {
        fprintf(stderr, "C_Sign: rv = 0x%.8X\n", rv);
        error = 1;
        goto exit_session;
    }

    fprintf(stdout, "Message was successfully signed with private key!\n");

    rv = C_VerifyInit(hSession, &smech, publickey);

    if (rv != CKR_OK) {
        fprintf(stderr, "C_VerifyInit: rv = 0x%.8X\n", rv);
        error = 1;
        goto exit_session;
    }

    rv = C_Verify(hSession, (CK_BYTE_PTR)message, messagelen,
                  (CK_BYTE_PTR)sign, slen);

    if (rv != CKR_OK) {
        fprintf(stderr, "C_Verify: rv = 0x%.8X\n", rv);
        error = 1;
        goto exit_session;
    }

    fprintf(stdout, "Message was successfully verified with public key!\n");

exit_session:
    (void) C_CloseSession(hSession);

exit_program:
    (void) C_Finalize(NULL_PTR);

    for (i = 0; i < template_size; i++) {
        if (getattributes[i].pValue != NULL)
            free(getattributes[i].pValue);
    }

    exit(error);

/* Find a slot capable of:
   * . signing and verifying with sv_mech
```
EXAMPLE 9–3  Signing and Verifying Text Using PKCS#11 Functions  

* generating a key pair with kpgen mech
* Returns B_TRUE when successful. */
boolean_t GetMySlot(CK_MECHANISM_TYPE sv_mech, CK_MECHANISM_TYPE kpgen_mech,
CK_SLOT_ID_PTR pSlotID)
{
  CK_SLOT_ID_PTR pSlotList = NULL_PTR;
  CK_SLOT_ID SlotID;
  CK_UULONG ulSlotCount = 0;
  CK_MECHANISM_INFO mech_info;
  int i;
  boolean_t returnval = B_FALSE;
  CK_RV rv;

  /* Get slot list for memory alloction */
  rv = C_GetSlotList(0, NULL_PTR, &ulSlotCount);
  if ((rv == CKR_OK) && (ulSlotCount > 0)) {
    fprintf(stdout, "slotCount = %d\n", ulSlotCount);
    pSlotList = malloc(ulSlotCount * sizeof (CK_SLOT_ID));
    if (pSlotList == NULL) {
      fprintf(stderr, "System error: unable to allocate "
                 "memory\n");
      return (returnval);
    }
    /* Get the slot list for processing */
    rv = C_GetSlotList(0, pSlotList, &ulSlotCount);
    if (rv != CKR_OK) {
      fprintf(stderr, "GetSlotList failed: unable to get "
                 "slot count.\n");
      goto cleanup;
    }
  } else {
    fprintf(stderr, "GetSlotList failed: unable to get slot "
               "list.\n");
    return (returnval);
  }

  /* Find a slot capable of specified mechanism */
  for (i = 0; i < ulSlotCount; i++) {
    SlotID = pSlotList[i];

    /* Check if this slot is capable of signing and */
    /* verifying with sv_mech. */
    rv = C_GetMechanismInfo(SlotID, sv_mech, &mech_info);
    if (rv != CKR_OK) {
      continue;
    }
    if (!(mech_info.flags & CKF_SIGN &&
          mech_info.flags & CKF_VERIFY)) {
      continue;
    }
    returnval = B_TRUE;
  }
  return (returnval);
}

User-Level Cryptographic Application Examples
/* Check if the slot is capable of key pair generation */
rv = C_GetMechanismInfo(SlotID, kpgen_mech, &mech_info);
if (rv != CKR_OK) {
    continue;
}
if (!(mech_info.flags & CKF_GENERATE_KEYPAIR)) {
    continue;
}

/* If we get this far, this slot supports our mechanisms. */
returnval = B_TRUE;
*pSlotID = SlotID;
break;
}
cleanup:
    if (pSlotList)
        free(pSlotList);
    return (returnval);

Random Byte Generation Example

Example 9–4 demonstrates how to find a slot with a mechanism that can generate random bytes. The example performs the following steps:

1. Initializes the cryptoki library.
2. Calls GetRandSlot() to find a slot with a mechanism that can generate random bytes.

The task of finding a slot performs the following steps:

a. Calling the function C_GetSlotList() to get a list of the available slots.
   C_GetSlotList() is called twice, as the PKCS #11 convention suggests.
   C_GetSlotList() is called the first time to get the number of slots for memory allocation. C_GetSlotList() is called the second time to retrieve the slots.

b. Finding a slot that can generate random bytes.
   For each slot, the function obtains the token information by using GetTokenInfo() and checks for a match with the CKF_RNG flag set. When a slot that has the CKF_RNG flag set is found, the GetRandSlot() function returns.

3. Opens the session by using C_OpenSession().
4. Generates random bytes by using C_GenerateRandom().
5. Ends the session.
The program uses `C_CloseSession()` to close the session and `C_Finalize()` to close the library.

The source code for the random number generation sample is shown in the following example.

**EXAMPLE 9–4 Generating Random Numbers Using PKCS#11 Functions**

```c
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <security/cryptoki.h>
#include <security/pkcs11.h>

#define RANDSIZE 64

boolean_t GetRandSlot(CK_SLOT_ID_PTR pslot);

/* Example generates random bytes. */
void main(int argc, char **argv) {
    CK_RV rv;
    CK_MECHANISM mech;
    CK_SESSION_HANDLE hSession;
    CK_SESSION_INFO sessInfo;
    CK_SLOT_ID slotID;
    CK_BYTE randBytes[RANDSIZE];

    boolean_t found_slot = B_FALSE;
    int error;
    int i;

    /* Initialize the CRYPTOKI library */
    rv = C_Initialize(NULL_PTR);
    if (rv != CKR_OK) {
        fprintf(stderr, "C_Initialize: Error = 0x%.8X\n", rv);
        exit(1);
    }

    found_slot = GetRandSlot(&slotID);
    if (!found_slot) {
        goto exit_program;
    }

    /* Open a session on the slot found */
    rv = C_OpenSession(slotID, CKF_SERIAL_SESSION, NULL_PTR, NULL_PTR, &hSession);
    if (rv != CKR_OK) {
        fprintf(stderr, "C_OpenSession: rv = 0x%.8x\n", rv);
        error = 1;
        goto exit_program;
    }

    /* Generate random bytes */
```
EXAMPLE 9–4  Generating Random Numbers Using PKCS #11 Functions  (Continued)

rv = C_GenerateRandom(hSession, randBytes, RANDSIZE);
if (rv != CKR_OK) {
    fprintf(stderr, "C_GenerateRandom: rv = 0x%.8x\n", rv);
    error = 1;
    goto exit_session;
}

fprintf(stdout, "Random value: ");
for (i = 0; i < RANDSIZE; i++) {
    fprintf(stdout, "%.2x", randBytes[i]);
}

exit_session:
    (void) C_CloseSession(hSession);

exit_program:
    (void) C_Finalize(NULL_PTR);
    exit(error);

boolean_t
GetRandSlot(CK_SLOT_ID_PTR pslot)
{
    CK_SLOT_ID_PTR pSlotList;
    CK_SLOT_ID SlotID;
    CK_TOKEN_INFO tokenInfo;
    CK_ULONG ulSlotCount;
    CK_MECHANISM_TYPE_PTR pMechTypeList = NULL_PTR;
    CK_ULONG ulMechTypecount;
    boolean_t result = B_FALSE;
    int i = 0;
    CK_RV rv;

    /* Get slot list for memory allocation */
    rv = C_GetSlotList(0, NULL_PTR, &ulSlotCount);
    if ((rv == CKR_OK) && (ulSlotCount > 0)) {
        fprintf(stdout, "slotCount = %d\n", (int)ulSlotCount);
        pSlotList = malloc(ulSlotCount * sizeof (CK_SLOT_ID));
        if (pSlotList == NULL) {
            fprintf(stderr, "System error: unable to allocate memory\n");
            return (result);
        }

        /* Get the slot list for processing */
        rv = C_GetSlotList(0, pSlotList, &ulSlotCount);
        if (rv == CKR_OK) {
            fprintf(stderr, "GetSlotList failed: unable to get "
                "slot list.\n");
            free(pSlotList);
            return (result);
        }
    }
EXAMPLE 9–4  Generating Random Numbers Using PKCS #11 Functions  

(Continued)

)}
    else {
        fprintf(stderr, "GetSlotList failed: unable to get slot" 
            "count.\n"); 
        return (result);
    }

/* Find a slot capable of doing random number generation */
for (i = 0; i < ulSlotCount; i++) {
    SlotID = pSlotList[i];
    rv = C_GetTokenInfo(SlotID, &tokenInfo);
    if (rv != CKR_OK) {
        /* Check the next slot */
        continue;
    }
    if (tokenInfo.flags & CKF_RNG) {
        /* Found a random number generator */
        *pslot = SlotID;
        fprintf(stderr, "Slot # %d supports random number " 
            "generation!\n", SlotID);
        result = B_TRUE;
        break;
    }
}

if (pSlotList)
    free(pSlotList);

return (result);
A smart card is a portable computer with a microprocessor and memory. A smart card usually has the shape and size of a credit card. Smart cards provide highly secure storage for confidential information that can be protected through authentication and encryption.

This chapter covers the following topics:
- “Oracle Solaris Smart Card Framework Overview” on page 181
- “Developing Smart Card Consumer Applications” on page 182
- “Developing an IFD Handler for Smart Card Terminals” on page 185
- “Installation of Smart Card Terminals” on page 186

**Oracle Solaris Smart Card Framework Overview**

In the Oracle Solaris operating system, the smart card framework is used to connect consumer applications with smart card terminals. A consumer application makes calls to the smart card framework (SCF) API. A smart card terminal communicates with consumer applications through an interface device (IFD) handler, which is basically a device driver. IFD handlers connect to the framework through the terminal interface. See the following figure.
The Oracle Solaris operating system stores smart card configuration information in a private file. This approach is in contrast to Linux implementations, where `/etc/reader.conf` is generally used. To change entries in the configuration file, use the command `smartcard(1M)`.

At this time, the smart card framework is independent of the Oracle Solaris cryptographic framework.

Developing Smart Card Consumer Applications

The SCF API provides a set of interfaces for accessing smart cards. These interfaces provide communication to the cards in low-level application protocol data unit (APDU) form. These interfaces are provided in both C and Java. The interfaces work with all readers that are supported by the Solaris operating system and with any smart card that communicates with APDUs. The SCF API is based on the following components:

- **Session object** – A general context for each individual thread so that can collisions can be avoided.
- **Terminal object** – An abstraction of a physical smart card terminal. This object can detect the presence, insertion, or removal of a card.
- **Card object** – Represents a smart card that is inserted in a terminal. The object can send information in APDU format to the physical smart card. The object also accommodates mutex locking so that an application can have exclusive access to the card.
- **Listener object** – An object that receives notification of events.

The SCF API provides functionality in the following areas:

- Checking for the physical presence of a smart card in the reader.
- Receiving notification of smart card movement, that is, insertion and removal.
• Exchanging data with the smart card.
• Retrieving information about the session, terminal, and smart card.
• Locking and unlocking the smart card for exclusive access.

The following sections provide information about the specific SCF interfaces.

**SCF Session Interfaces**

The following functions are used for SCF sessions.

**SCF_Session_getSession**

Establishes a session with a system’s smart card framework. After a session has been opened, the session can be used with `SCF_Session_getTerminal` to access a smart card terminal.

**SCF_Session_close**

Releases the resources that were allocated when the session was opened. Also, closes any terminals or cards that are associated with that session.

**SCF_Session_getInfo**

Obtains information about a session.

**SCF_Session_freeInfo**

Deallocates storage that is returned from `SCF_Session_getInfo`.

**SCF_Session_getTerminal**

Establishes a context with a specific smart card terminal in the session. Terminal objects are used for detecting card movement, that is, insertion or removal. Terminal objects are also used to create card objects for accessing a specific card.

**SCF Terminal Interfaces**

The following functions are used to access SCF terminals.

**SCF_Terminal_close**

Releases the resources that were allocated when the terminal was opened. The function also closes any cards that were associated with the terminal.

**SCF_Terminal_getInfo**

Obtains information about a terminal.

**SCF_Terminal_freeInfo**

Deallocates storage that has been returned from `SCF_Terminal_getInfo`.

**SCF_Terminal_waitForCardPresent**

Blocks and waits until a card is present in the specified terminal.
SCF_Terminal_waitForCardAbsent(3SMARTCARD)
Blocks and waits until the card in the specified terminal is removed.

SCF_Terminal_addEventListener(3SMARTCARD)
Allows a program to receive callback notifications when events occur on a terminal. The concept is similar to a signal handler. When an event occurs, a service thread executes the provided callback function.

SCF_Terminal_updateEventListener(3SMARTCARD)
Updates the specified event listener that is associated with this terminal.

SCF_Terminal_removeEventListener(3SMARTCARD)
Removes the specified event listener from the listener list that is associated with this terminal.

SCF_Terminal_getCard(3SMARTCARD)
Establishes a context with a specific smart card in a terminal. Card objects can be used to send APDUs to the card with SCF_Card_exchangeAPDU(3SMARTCARD).

SCF Card and Miscellaneous Interfaces

The following functions are used to access smart cards and to get status.

SCF_Card_close(3SMARTCARD)
Releases resources, such as memory and threads, that were allocated when the card was opened. Also, releases the lock that was held by that card.

SCF_Card_getInfo(3SMARTCARD)
Obtains information about a card.

SCF_Card_freeInfo(3SMARTCARD)
Deallocates storage that has been returned from SCF_Card_getInfo(3SMARTCARD).

SCF_Card_lock(3SMARTCARD)
Obtains a lock on a specific card. This function allows an application to perform a multiple APDU transaction without interference from other smart card applications.

SCF_Card_unlock(3SMARTCARD)
Removes a lock from a specific card.

SCF_Card_exchangeAPDU(3SMARTCARD)
Sends a command APDU to a card and reads the card's response.

SCF_Card_waitForCardRemoved(3SMARTCARD)
Checks to see if a specific card has been removed. If another card or even the same card has since been reinserted, the function reports that the old card was removed.

SCF_Card_reset(3SMARTCARD)
Resets a specific card.
Developing an IFD Handler for Smart Card Terminals

Smart card terminals that are developed for the Oracle Solaris OS use the same set of APIs that are used by Linux smart card terminals. If you have not previously developed an IFD handler, then you should visit one of the websites for the Linux environment that provide IFD source code, such as [http://www.musclecard.com/drivers.html](http://www.musclecard.com/drivers.html). To develop an IFD handler for smart card terminals in the Solaris operating system, you need to include `/usr/include/smartcard/ifdhandler.h` and implement the following interfaces:

- **IFDCreateChannelByName(3SMARTCARD)** – Opens a communication channel with the specified smart card terminal. This interface is new in the latest version of the MUSCLE IFD specification. As a result, `IFDCreateChannelByName()` might not be available in other IFD handlers. In the Solaris software, `IFDCreateChannelByName()` is used instead of the `IFDCreateChannel(3SMARTCARD)` function.
- **IFDHICCPresence(3SMARTCARD)** – Checks for the presence of an ICC, that is, a smart card, in the reader or the slot that has been specified by the logical unit number (LUN).
- **IFDHPowerICC(3SMARTCARD)** – Controls the power and reset signals of the ICC.
- **IFDHCloseChannel(3SMARTCARD)** – Closes the communications channel for the IFD that is specified by LUN.
- **IFDGetCapabilities(3SMARTCARD)** – Returns the capabilities of the specified smart card, IFD handler, or smart card terminal.
- **IFDSetProtocolParameters(3SMARTCARD)** – Sets the Protocol Type Selection (PTS) for a particular slot or card. Check the ISO 7816 standards for the PTS values. Although this function might not be called by the framework, this function should be implemented. Use of `IFDSetProtocolParameters()` ensures that a variety of cards can communicate with the framework.
- **IFDTransmitToICC(3SMARTCARD)** – Called by the framework to communicate with the smart card.

**Note** – The `IFDCreateChannel()`, `IFDGetCapabilities()`, and `IFDControl()` are not currently used, but these interfaces might be required in future releases.

The `IFDHICCPresence()` and `IFDHPowerICC()` function are useful for testing. For example, you can use the `IFDHICCPresence()` function to test the presence of a card in the slot. One way to check the smart card power is functioning normally is to use the `IFDHPowerICC()` function. This function gets the Answer to Reset (ATR) value of the inserted smart card.
Installation of Smart Card Terminals

The Solaris smart card framework does not support hot-pluggable terminals, such as USB terminals. Use the following approach to connect and install smart card terminals:

1. Make the physical connection of the terminal to the system.
2. Copy the shared library for the IFD handler to the system.
3. Register the IFD handler for the terminal into the framework with `smartcard(1M)`. 
This appendix shows the source code for two sample applications that use GSS-API to make a safe network connection. The first application is a typical client. The second application demonstrates how a server works in GSS-API. The two programs display benchmarks in the course of being run. A user can thus view GSS-API in action. Additionally, certain miscellaneous functions are provided for use by the client and server applications.

- “Client-Side Application” on page 187
- “Server-Side Application” on page 197
- “Miscellaneous GSS-API Sample Functions” on page 207

These programs are examined in detail in the Chapter 5, "GSS-API Client Example," and Chapter 6, "GSS-API Server Example."

### Client-Side Application

The source code for the client-side program, gss_client, is provided in the following example.

**EXAMPLE A-1  Complete Listing of gss-client.c Sample Program**

```c
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
```
EXAMPLE A-1 Complete Listing of gss-client.c Sample Program (Continued)

* CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
* USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
* OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
* PERFORMANCE OF THIS SOFTWARE.
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include <gss-misc.h>

void usage()
{
    fprintf(stderr, "Usage: gss-client [-port port] [-d] host service \ 
msg\n";
    exit(1);
}

/*
* Function: connect_to_server
* Purpose: Opens a TCP connection to the name host and port.
* Arguments:
* host (r) the target host name
* port (r) the target port, in host byte order
* Returns: the established socket file descriptor, or -1 on failure
* Effects:
* The host name is resolved with gethostbyname(), and the socket is
* opened and connected. If an error occurs, an error message is
* displayed and -1 is returned.
*/
int connect_to_server(host, port)
    char *host;
EXAMPLE A–1  Complete Listing of gss-client.c Sample Program  (Continued)

    u_short port;
    {
        struct sockaddr_in saddr;
        struct hostent *hp;
        int s;

        if ((hp = gethostbyname(host)) == NULL) {
            fprintf(stderr, "Unknown host: %s\n", host);
            return -1;
        }

        saddr.sin_family = hp->h_addrtype;
        memcpy((char *)&saddr.sin_addr, hp->h_addr, sizeof(saddr.sin_addr));
        saddr.sin_port = htons(port);

        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("creating socket");
            return -1;
        }

        if (connect(s, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
            perror("connecting to server");
            (void) close(s);
            return -1;
        }

        return s;
    }

/*
 * Function: client_establish_context
 * Purpose: establishes a GSS-API context with a specified service and
 * returns the context handle
 * Arguments:
 *    s (r) an established TCP connection to the service
 *    service_name (r) the ASCII service name of the service
 *    context (w) the established GSS-API context
 *    ret_flags (w) the returned flags from init_sec_context
 * Returns: 0 on success, -1 on failure
 * Effects:
 *    service_name is imported as a GSS-API name and a GSS-API context is
 *    established with the corresponding service; the service should be
 *    listening on the TCP connection s. The default GSS-API mechanism
 *    is used, and mutual authentication and replay detection are
 *    requested.
 *    If successful, the context handle is returned in context. If
 *    unsuccessful, the GSS-API error messages are displayed on stderr
 *    and -1 is returned.
 */
int client_establish_context(s, service_name, deleg_flag, oid,
    gss_context, ret_flags)
int s;
char *service_name;
gss OID oid;
OM uint32 deleg_flag;
gss_ctx_id_t *gss_context;
OM uint32 *ret_flags;
{
gss_buffer_desc send_tok, recv_tok, *token_ptr;
gss_name_t target_name;
OM_uint32 maj_stat, min_stat, init_sec_min_stat;

/*
 * Import the name into target_name. Use send_tok to save
 * local variable space.
 */
send_tok.value = service_name;
send_tok.length = strlen(service_name) + 1;
maj_stat = gss_import_name(&min_stat, &send_tok,
    (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("parsing name", maj_stat, min_stat);
    return -1;
}

/*
 * Perform the context-establishment loop.
 *
 * On each pass through the loop, token_ptr points to the token
 * to send to the server (or GSS_C_NO BUFFER on the first pass).
 * Every generated token is stored in send_tok which is then
 * transmitted to the server; every received token is stored in
 * recv_tok, which token_ptr is then set to, to be processed by
 * the next call to gss_init_sec_context.
 *
 * GSS-API guarantees that send_tok's length will be non-zero
 * if and only if the server is expecting another token from us,
 * and that gss_init_sec_context returns GSS_S_CONTINUE_NEEDED if
 * and only if the server has another token to send us.
 */

token_ptr = GSS_C_NO_BUFFER;
*gss_context = GSS_C_NO_CONTEXT;

do {
    maj_stat =
        gss_init_sec_context(&init_sec_min_stat,
            GSS_C_NO_CREDENTIAL,
            gss_context,
            target_name,
            oid,
            GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG |
                deleg_flag,
            0,
            NULL, /* no channel bindings */
            token_ptr,
            NULL, /* ignore mech type */
if (token_ptr != GSS_C_NO_BUFFER) {
    (void) gss_release_buffer(&min_stat, &recv_tok);
    if (send_tok.length != 0) {
        printf("Sending init_sec_context token (size=%d)...", 
            send_tok.length);
        if (send_token(s, &send_tok) < 0) {
            (void) gss_release_buffer(&min_stat, &send_tok);
            (void) gss_release_name(&min_stat, &target_name);
            return -1;
        }
    }
    (void) gss_release_buffer(&min_stat, &send_tok);
    if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
        display_status("initializing context", maj_stat, 
            init_sec_min_stat);
        (void) gss_release_name(&min_stat, &target_name);
        if (*gss_context == GSS_C_NO_CONTEXT)
            gss_delete_sec_context(&min_stat, gss_context, 
                GSS_C_NO_BUFFER);
        return -1;
    }
}
if (maj_stat == GSS_S_CONTINUE_NEEDED) {
    printf("continue needed...");
    if (recv_token(s, &recv_tok) < 0) {
        (void) gss_release_name(&min_stat, &target_name);
        return -1;
    } else if (recv_tok != GSS_C_NO_TOKEN) {
        token_ptr = &recv_tok;
        printf("\n");
    }
}
while (maj_stat == GSS_S_CONTINUE_NEEDED);
(void) gss_release_name(&min_stat, &target_name);
return 0;

void read_file(file_name, in_buf)
    char *file_name;
    gss_buffer_t in_buf;
{
    int fd, bytes_in, count;
    struct stat stat_buf;
    if ((fd = open(file_name, O_RDONLY, 0)) < 0) {
        perror("open");
        fprintf(stderr, "Couldn’t open file %s\n", file_name);
        exit(1);
    }
    if (fstat(fd, &stat_buf) < 0) {
EXAMPLE A–1  Complete Listing of gss-client.c Sample Program  (Continued)

```c
    perror("fstat");
    exit(1);
}

in_buf->length = stat_buf.st_size;

if (in_buf->length == 0) {
    in_buf->value = NULL;
    return;
}

if ((in_buf->value = malloc(in_buf->length)) == 0) {
    fprintf(stderr,
            "Couldn't allocate %d byte buffer for reading file\n",
            in_buf->length);
    exit(1);
}

/* this code used to check for incomplete reads, but you can't get
   an incomplete read on any file for which fstat() is meaningful */

count = read(fd, in_buf->value, in_buf->length);
if (count < 0) {
    perror("read");
    exit(1);
}

if (count < in_buf->length)
    fprintf(stderr, "Warning, only read in %d bytes, expected %d\n",
            count, in_buf->length);
}

/*
 * Function: call_server
 * Purpose: Call the "sign" service.
 * Arguments:
 * host (r) the host providing the service
 * port (r) the port to connect to on host
 * service_name (r) the GSS-API service name to authenticate to
 * msg (r) the message to have "signed"
 * Returns: 0 on success, -1 on failure
 * Effects:
 */

int call_server(host, port, oid, service_name, msg, use_file)
    char *host;
    u_short port;
    gss_OID oid;
    char *msg;
    char *service_name;
    int use_file;
    int deleg_flag;
```
EXAMPLE A–1  Complete Listing of gss-client.c Sample Program  (Continued)

    char *service_name;
    OM_uint32 deleg_flag;
    char *msg;
    int use_file;

    {
        gss_ctx_id_t context;
        gss_buffer_desc in_buf, out_buf;
        int s, state;
        OM_uint32 ret_flags;
        OM_uint32 maj_stat, min_stat;
        gss_name_t src_name, targ_name;
        gss_buffer_desc sname, tname;
        OM_uint32 lifetime;
        gss_OID mechanism, name_type;
        int is_local;
        OM_uint32 context_flags;
        int is_open;
        gss_qop_t qop_state;
        gss_OID_set mech_names;
        gss_buffer_desc oid_name;
        size_t i;

        /* Open connection */
        if ((s = connect_to_server(host, port)) < 0)
            return -1;

        /* Establish context */
        if (client_establish_context(s, service_name, deleg_flag, oid,
                &context, &ret_flags) < 0) {
            (void) close(s);
            return -1;
        }

        /* display the flags */
        display_ctx_flags(ret_flags);

        /* Get context information */
        maj_stat = gss_inquire_context(&min_stat, context,
                &src_name, &targ_name, &lifetime,
                &mechanism, &context_flags,
                &is_local,
                &is_open);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("inquiring context", maj_stat, min_stat);
            return -1;
        }

        maj_stat = gss_display_name(&min_stat, src_name, &sname,
                &name_type);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("displaying source name", maj_stat, min_stat);
            return -1;
        }

        maj_stat = gss_display_name(&min_stat, targ_name, &tname,
                (gss_OID *) NULL);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("displaying target name", maj_stat, min_stat);
            return -1;
        }

        /* Optional I/O */
        for(i = 0; i < num_requests; i++) {
            /* Do something */
        }

        /* Finalize */
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("finalizing context", maj_stat, min_stat);
            return -1;
        }
    }

    /* Close connection */
    close(s);
EXAMPLE A-1  Complete Listing of gss-client.c Sample Program  (Continued)

        display_status("displaying target name", maj_stat, min_stat);
        return -1;
    }
    fprintf(stderr, "%.*s\tto \%.*s\n", lifetime %d, flags %x, %s,
             %s
         (int) sname.length, (char *) sname.value,
         (int) tname.length, (char *) tname.value, lifetime,
         context flags,
         (is_local) ? "locally initiated" : "remotely initiated",
         (is_open) ? "open" : "closed",
         (is_local) ? "locally initiated" : "remotely initiated",
         (is_open) ? "open" : "closed";
    (void) gss_release_name(&min_stat, &src_name);
    (void) gss_release_name(&min_stat, &targ_name);
    (void) gss_release_buffer(&min_stat, &sname);
    (void) gss_release_buffer(&min_stat, &tname);
    maj_stat = gss_oid_to_str(&min_stat,
                   name_type,
                   &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    fprintf(stderr, "Name type of source name is %.*s\n",
             (int) oid_name.length, (char *) oid_name.value);
    (void) gss_release_buffer(&min_stat, &oid_name);

/* Now get the names supported by the mechanism */
    maj_stat = gss_inquire_names_for_mech(&min_stat,
                   mechanism,
                   &mech_names);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("inquiring mech names", maj_stat, min_stat);
        return -1;
    }
    maj_stat = gss_oid_to_str(&min_stat,
                   mechanism,
                   &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    fprintf(stderr, "Mechanism %.*s supports %d names\n",
             (int) oid_name.length, (char *) oid_name.value,
             mech_names->count);
    (void) gss_release_buffer(&min_stat, &oid_name);

    for (i=0; i<mech_names->count; i++) {
        maj_stat = gss_oid_to_str(&min_stat,
                         &mech_names->elements[i],
                         &oid_name);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("converting oid->string", maj_stat, min_stat);
            return -1;
        }
        fprintf(stderr, "%d: %.*s\n", i,

Client-Side Application

Developer's Guide to Oracle Solaris 10 Security • September 2013
(int) oid_name.length, (char *) oid_name.value);

(void) gss_release_buffer(&min_stat, &oid_name);
}
(void) gss_release_oid_set(&min_stat, &mech_names);

if (use_file) {
  read_file(msg, &in_buf);
} else {
  /* Seal the message */
  in_buf.value = msg;
  in_buf.length = strlen(msg);
}

maj_stat = gss_wrap(&min_stat, context, 1, GSS_C_QOP_DEFAULT,
                   &in_buf, &state, &out_buf);
if (maj_stat != GSS_S_COMPLETE) {
  display_status("sealing message", maj_stat, min_stat);
  (void) close(s);
  (void) gss_delete_sec_context(&min_stat, &context,
                                GSS_C_NO_BUFFER);
  return -1;
} else if (! state) {
  fprintf(stderr, "Warning! Message not encrypted.\n");
}

/* Send to server */
if (send_token(s, &out_buf) < 0) {
  (void) close(s);
  (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
  return -1;
}
(void) gss_release_buffer(&min_stat, &out_buf);

/* Read signature block into out_buf */
if (recv_token(s, &out_buf) < 0) {
  (void) close(s);
  (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
  return -1;
}

/* Verify signature block */
maj_stat = gss_verify_mic(&min_stat, context, &in_buf,
                         &out_buf, &qop_state);
if (maj_stat != GSS_S_COMPLETE) {
  display_status("verifying signature", maj_stat, min_stat);
  (void) close(s);
  (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
  return -1;
}
(void) gss_release_buffer(&min_stat, &out_buf);

if (use_file)
  free(in_buf.value);
printf("Signature verified.\n");
/ * Delete context */
  maj_stat = gss_delete_sec_context(&min_stat, &context, &out_buf);
  if (maj_stat != GSS_S_COMPLETE) {
    display_status("deleting context", maj_stat, min_stat);
    (void) close(s);
    (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
    return -1;
  }
  (void) gss_release_buffer(&min_stat, &out_buf);
  (void) close(s);
  return 0;
}

static void parse_oid(char *mechanism, gss_OID *oid)
{
  char *mechstr = 0, *cp;
  gss_buffer_desc tok;
  OM_uint32 maj_stat, min_stat;
  if (isdigit(mechanism[0])) {
    mechstr = malloc(strlen(mechanism)+5);
    if (!mechstr) {
      printf("Couldn't allocate mechanism scratch!\n");
      return;
    }
    sprintf(mechstr, "{% s}\n", mechanism);
    for (cp = mechstr; *cp; cp++)
      if (*cp == '.')
        *cp = ' ';
    tok.value = mechstr;
  } else
    tok.value = mechanism;
  tok.length = strlen(tok.value);
  maj_stat = gss_str_to_oid(&min_stat, &tok, oid);
  if (maj_stat != GSS_S_COMPLETE) {
    display_status("str_to_oid", maj_stat, min_stat);
    return;
  }
  if (mechstr)
    free(mechstr);
}

int main(argc, argv)
{
  int argc;
  char **argv;
  ...
}
/* Parse arguments. */
argc--; argv++;
while (argc) {
    if (strcmp(*argv, "-port") == 0) {
        argc--; argv++;
        port = atoi(*argv);
    } else if (strcmp(*argv, "-mech") == 0) {
        argc--; argv++;
        mechanism = *argv;
    } else if (strcmp(*argv, "-d") == 0) {
        deleg_flag = GSS_C_DELEG_FLAG;
    } else if (strcmp(*argv, "-f") == 0) {
        use_file = 1;
    } else
        break;
    argc--; argv++;
}
if (argc != 3)
    usage();
server_host = *argv++;
service_name = *argv++;
msg = *argv++;
if (mechanism)
    parse_oid(mechanism, &oid);
if (call_server(server_host, port, oid, service_name,
        deleg_flag, msg, use_file) < 0)
    exit(1);
if (oid != GSS_C_NULL_OID)
    (void) gss_release_oid(&min_stat, &oid);
return 0;

---

**Server-Side Application**

The source code for the server-side program, gss_server, is provided in the following example.

---

*Copyright 1994 by OpenVision Technologies, Inc.*

*Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in*
* supporting documentation, and that the name of OpenVision not be used
* in advertising or publicity pertaining to distribution of the software
* without specific, written prior permission. OpenVision makes no
* representations about the suitability of this software for any
* purpose. It is provided "as is" without express or implied warranty.
* 
* OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
* INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
* EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
* CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
* USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
* OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
* PERFORMANCE OF THIS SOFTWARE.
*/

#else if (!defined(lint) && !defined(__CODECENTER__))
static char *rcsid = \
"$Header: /cvs/krbdev/krb5/src/appl/gss-sample/gss-server.c, \
v 1.21 1998/12/22 \
04:10:08 tytso Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <ctype.h>
#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include <gss-misc.h>
#include <string.h>

void usage()
{
    fprintf(stderr, "Usage: gss-server [-port port] [-verbose]
"
    fprintf(stderr, " [-inetd] [-logfile file] [service_name]\n";
    exit(1);
}

FILE *log;

int verbose = 0;
/
* Function: server_acquire_creds
* Purpose: imports a service name and acquires credentials for it
* Arguments:
* service_name (r) the ASCII service name
* server_creds (w) the GSS-API service credentials
* * Returns: 0 on success, -1 on failure
* * Effects:
* * The service name is imported with gss_import_name, and service
* * credentials are acquired with gss_acquire_cred. If either operation
* * fails, an error message is displayed and -1 is returned; otherwise,
* * 0 is returned.
*/
int server_acquire_creds(char *service_name, gss_cred_id_t *server_creds)
{
   gss_buffer_desc name_buf;
   gss_name_t server_name;
   OM_uint32 maj_stat, min_stat;

   name_buf.value = service_name;
   name_buf.length = strlen(name_buf.value) + 1;
   maj_stat = gss_import_name(&min_stat, &name_buf,
                              (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
   if (maj_stat != GSS_S_COMPLETE) {
      display_status("importing name", maj_stat, min_stat);
      return -1;
   }

   maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                               (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, GSS_C_ACCEPT,
                               server_creds, NULL, NULL);
   if (maj_stat != GSS_S_COMPLETE) {
      display_status("acquiring credentials", maj_stat, min_stat);
      return -1;
   }

   (void) gss_release_name(&min_stat, &server_name);
   return 0;
}

/*
Function: server_establish_context
*
Purpose: establishes a GSS-API context as a specified service with
* an incoming client, and returns the context handle and associated
* client name
*
Arguments:
*
* s (r) an established TCP connection to the client
* service_creds (r) server credentials, from gss_acquire_cred
* context (w) the established GSS-API context
* client_name (w) the client's ASCII name
*
Returns: 0 on success, -1 on failure
EXAMPLE A-2  Complete Code Listing for gss-server.c Sample Program  (Continued)

*    * Effects:
    *    * Any valid client request is accepted. If a context is established,
    *    * its handle is returned in context and the client name is returned
    * in client_name and 0 is returned. If unsuccessful, an error
    * message is displayed and -1 is returned.
    */
int server_establish_context(s, server_creds, context, client_name, \ ret_flags)
{
    int s;
    gss_cred_id_t server_creds;
    gss_ctx_id_t *context;
    gss_buffer_t client_name;
    OM_uint32 *ret_flags;
{
    gss_buffer_desc send_tok, recv_tok;
    gss_name_t client;
    gss_OID doid;
    OM_uint32 maj_stat, min_stat, acc_sec_min_stat;
    gss_buffer_desc oid_name;
    *context = GSS_C_NO_CONTEXT;
    do {
        if (recv_token(s, &recv_tok) < 0)
            return -1;
        if (verbose && log) {
            fprintf(log, "Received token (size=%d): \n", recv_tok.length);
            print_token(&recv_tok);
        }
        maj_stat =
        gss_accept_sec_context(&acc_sec_min_stat,
            context,
            server_creds,
            &recv_tok,
            GSS_C_NO_CHANNEL_BINDINGS,
            &client,
            &doid,
            &send_tok,
            ret_flags,
            NULL, /* ignore time_rec */
            NULL);  /* ignore del_cred_handle */
        (void) gss_release_buffer(&min_stat, &recv_tok);
        if (send_tok.length != 0) {
            if (verbose && log) {
                fprintf(log,
                    "Sending accept_sec_context token (size=%d):\n", send_tok.length);
                print_token(&send_tok);
            }
        }
    }
EXAMPLE A–2  Complete Code Listing for gss-server.c Sample Program  (Continued)

    if (send_token(s, &send_tok) < 0) {
        fprintf(log, "failure sending token\n");
        return -1;
    }

    (void) gss_release_buffer(&min_stat, &send_tok);
}

if (maj_stat != GSS_S_COMPLETE & maj_stat != GSS_S_CONTINUE_NEEDED) {
    display_status("accepting context", maj_stat,
                   acc_sec_min_stat);
    if (*context == GSS_C_NO_CONTEXT)
        gss_delete_sec_context(&min_stat, context,
                               GSS_C_NO_BUFFER);
    return -1;
}

if (verbose && log) {
    if (maj_stat == GSS_S_CONTINUE_NEEDED)
        fprintf(log, "continue needed...\n");
    else
        fprintf(log, "\n");
    fflush(log);
}
}

while (maj_stat == GSS_S_CONTINUE_NEEDED);

/* display the flags */
display_ctx_flags(*ret_flags);

if (verbose && log) {
    maj_stat = gss_oid_to_str(&min_stat, doid, &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
    fprintf(log, "Accepted connection using mechanism OID %.*s.\n",
            (int) oid_name.length, (char *) oid_name.value);
    (void) gss_release_buffer(&min_stat, &oid_name);
}

maj_stat = gss_display_name(&min_stat, client, client_name, &doid);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("displaying name", maj_stat, min_stat);
    return -1;
}

maj_stat = gss_release_name(&min_stat, &client);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("releasing name", maj_stat, min_stat);
    return -1;
}

return 0;

/*
 * Function: create_socket
 *
 * Purpose: Opens a listening TCP socket.
 */
EXAMPLE A-2  Complete Code Listing for gss-server.c Sample Program  (Continued)

* Arguments:
  * port        (r) the port number on which to listen
* Returns: the listening socket file descriptor, or -1 on failure
* Effects:
  * A listening socket on the specified port is created and returned.
  * On error, an error message is displayed and -1 is returned.
*/
int create_socket(port)
  u_short port;
{
  struct sockaddr_in saddr;
  int s;
  int on = 1;

  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(port);
  saddr.sin_addr.s_addr = INADDR_ANY;

  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("creating socket");
      return -1;
  }
  /* Let the socket be reused right away */
  void setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&on,
                  sizeof(on));
  if (bind(s, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {
      perror("binding socket");
      (void) close(s);
      return -1;
  }
  if (listen(s, 5) < 0) {
      perror("listening on socket");
      (void) close(s);
      return -1;
  }
  return s;
}

static float timeval_subtract(tv1, tv2)
  struct timeval *tv1, *tv2;
{
  return ((tv1->tv_sec - tv2->tv_sec) +
          ((float) (tv1->tv_usec - tv2->tv_usec)) / 1000000);
}

/* Yes, yes, this isn't the best place for doing this test.
* DO NOT REMOVE THIS UNTIL A BETTER TEST HAS BEEN WRITTEN, THOUGH.
  -TYT
*/
int test_import_export_context(context)
EXAMPLE A-2  Complete Code Listing for gss-server.c Sample Program  (Continued)

```c
    gss_ctx_id_t *context;
    {
        OM_uint32       min_stat, maj_stat;
        gss_buffer_desc context_token, copied_token;
        struct timeval tm1, tm2;

        /*
        * Attempt to save and then restore the context.
        */
        gettimeofday(&tm1, (struct timezone *)0);
        maj_stat = gss_export_sec_context(&min_stat, context, &context_token);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("exporting context", maj_stat, min_stat);
            return 1;
        }
        gettimeofday(&tm2, (struct timezone *)0);
        if (verbose && log)
            fprintf(log, "Exported context: %d bytes, %7.4f seconds\n",
                    context_token.length, timeval_subtract(&tm2, &tm1));
        copied_token.length = context_token.length;
        copied_token.value = malloc(context_token.length);
        if (copied_token.value == 0) {
            fprintf(log, "Couldn’t allocate memory to copy context token.\n");
            return 1;
        }
        memcpy(copied_token.value, context_token.value, copied_token.length);
        maj_stat = gss_import_sec_context(&min_stat, &copied_token, context);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("importing context", maj_stat, min_stat);
            return 1;
        }
        free(copied_token.value);
        gettimeofday(&tm1, (struct timezone *)0);
        if (verbose && log)
            fprintf(log, "Importing context: %7.4f seconds\n",
                    timeval_subtract(&tm1, &tm2));
    } (void) gss_release_buffer(&min_stat, &context_token);
    return 0;
}

/*
 * Function: sign_server
 * Purpose: Performs the "sign" service.
 * Arguments:
 *  s           (r) a TCP socket on which a connection has been accepted
 *  service_name (r) the ASCII name of the GSS-API service to establish a context as
 */
```
EXAMPLE A-2  Complete Code Listing for gss-server.c Sample Program  (Continued)

* Returns: -1 on error
* Effects:
* sign_server establishes a context, and performs a single sign request.
* A sign request is a single GSS-API sealed token. The token is
unsealed and a signature block, produced with gss_sign, is returned
* to the sender. The context is then destroyed and the connection
* closed.
* If any error occurs, -1 is returned.
*/
int sign_server(s, server_creds)
    int s;
    gss_cred_id_t server_creds;
{
    gss_buffer_desc client_name, xmit_buf, msg_buf;
    gss_ctx_id_t context;
    OM_uint32 maj_stat, min_stat;
    int i, conf_state, ret_flags;
    char *cp;

    /* Establish a context with the client */
    if (server_establish_context(s, server_creds, &context,
                                  &client_name, &ret_flags) < 0)
        return(-1);
    printf("Accepted connection: \%s\n", (int) client_name.length, (char *) client_name.value);
    (void) gss_release_buffer(&min_stat, &client_name);
    for (i=0; i<3 ; i++)
        if (test_import_export_context(&context))
            return -1;

    /* Receive the sealed message token */
    if (recv_token(s, &xmit_buf) < 0)
        return(-1);
    if (verbose && log) {
        fprintf(log, "Sealed message token:\n");
        print_token(&xmit_buf);
    }

    maj_stat = gss unwrap(&min_stat, context, &xmit_buf, &msg_buf,
                          &conf_state, (gss_qop_t *) NULL);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("unsealing message", maj_stat, min_stat);
        return(-1);
    } else if (! conf_state) {
        fprintf(stderr, "Warning! Message not encrypted.\n");
    }
    (void) gss_release_buffer(&min_stat, &xmit_buf);
Complete Code Listing for gss-server.c Sample Program

EXAMPLE A–2

```c
fprintf(log, "Received message: ");
cp = msg_buf.value;
if ((isprint(cp[0]) || isspace(cp[0])) &&
    (isprint(cp[1]) || isspace(cp[1]))) {
    fprintf(log, "\%.*s\n", msg_buf.length, msg_buf.value);
} else {
    printf("\n");
    print_token(&msg_buf);
}

/* Produce a signature block for the message */
maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT,
    &msg_buf, &xmit_buf);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("signing message", maj_stat, min_stat);
    return(-1);
}
(void) gss_release_buffer(&min_stat, &msg_buf);

/* Send the signature block to the client */
if (send_token(s, &xmit_buf) < 0)
    return(-1);
(void) gss_release_buffer(&min_stat, &xmit_buf);

/* Delete context */
maj_stat = gss_delete_sec_context(&min_stat, &context, NULL);
if (maj_stat != GSS_S_COMPLETE) {
    display_status("deleting context", maj_stat, min_stat);
    return(-1);
}
fflush(log);
return(0);
```

int main(argc, argv)
int argc;
char **argv;
{
    char *service_name;
gss_cred_id_t server_creds;
    OM_uint32 min_stat;
    u_short port = 4444;
    int s;
    int once = 0;
    int do_inetd = 0;

    log = stdout;
display_file = stdout;
argc--; argv++;
while (argc) {
    if (strcmp(*argv, "-port") == 0) {
        port = atoi(argv[1]);
    }
```
EXAMPLE A-2  Complete Code Listing for gss-server.c Sample Program  (Continued)

    argc--; argv++;  
    if (!argc) usage();  
    port = atoi(*argv);  
} else if (strcmp(*argv, "-verbose") == 0) {  
    verbose = 1;  
} else if (strcmp(*argv, "-once") == 0) {  
    once = 1;  
} else if (strcmp(*argv, "-inetd") == 0) {  
    do_inetd = 1;  
} else if (strcmp(*argv, "-logfile") == 0) {  
    argc--; argv++;  
    if (!argc) usage();  
    log = fopen(*argv, "a");  
    display_file = log;  
    if (!log) {  
        perror(*argv);  
        exit(1);  
    }  
} else  
    break; 
    argc--; argv++; 
}

if (argc != 1).usage(); 
if ((*argv)[0] == '-') usage(); 

service_name = *argv; 

if (server_acquire_creds(service_name, &server_creds) < 0) 
    return -1; 

if (do_inetd) {  
    close(1);  
    close(2);  
    sign_server(0, server_creds);  
    close(0);  
} else {  
    int stmp;  
    if ((stmp = create_socket(port)) >= 0) {  
        do {  
            /* Accept a TCP connection */  
            if ((s = accept(stmp, NULL, 0)) < 0) {  
                perror("accepting connection");  
                continue;  
            }  
            /* this return value is not checked, because there’s 
              not really anything to do if it fails */  
            sign_server(s, server_creds);  
            close(s);  
        } while (!once);  
        close(stmp);  
    }
EXAMPLE A–2  Complete Code Listing for gss-server.c Sample Program  (Continued)

    }
    }
    (void) gss_release_cred(&min_stat, &server_creds);
    /*NOTREACHED*/
    (void) close(s);
    return 0;

Miscellaneous GSS-API Sample Functions

To make the client and server programs work as shown, a number of other functions are required. These functions are used to display values. The functions are not otherwise needed. The functions in this category are as follows:

- send_token() – Transfers tokens and messages to a recipient
- recv_token() – Accepts tokens and messages from a sender
- display_status() – Shows the status returned by the last GSS-API function called
- write_all() – Writes a buffer to a file
- read_all() – Reads a file into a buffer
- display_ctx_flags() – Shows in a readable form information about the current context, such as whether confidentiality or mutual authentication is allowed
- print_token() – Prints out a token’s value

The code for these functions is shown in the following example.

EXAMPLE A–3  Code Listings for Miscellaneous GSS-API Functions

/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose. It is provided “as is” without express or implied warranty.
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
*/
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include <gss-misc.h>
#include <stdlib.h>

FILE *display_file;

static void display_status_1 (char *m, OM_uint32 code, int type);

static int write_all(int fildes, char *buf, unsigned int nbyte)
{
    int ret;
    char *ptr;
    for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
        ret = write(fildes, ptr, nbyte);
        if (ret < 0) {
            if (errno == EINTR)
                continue;
            return(ret);
        } else if (ret == 0) {
            return(ptr-buf);
        }
    }
    return(ptr-buf);
}

static int read_all(int fildes, char *buf, unsigned int nbyte)
{
    int ret;
    char *ptr;
    for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
        ret = read(fildes, ptr, nbyte);
        if (ret < 0) {
            if (errno == EINTR)
                continue;
            return(ret);
        } else if (ret == 0) {
            return(ptr-buf);
        }
    }
    return(ptr-buf);
}
return(ret);
    } else if (ret == 0) {
      return(ptr-buf);
    }
  }

  return(ptr-buf);
}

/*
 * Function: send_token
 * Purpose: Writes a token to a file descriptor.
 * Arguments:
 * s (r) an open file descriptor
 * tok (r) the token to write
 * Returns: 0 on success, -1 on failure
 * Effects:
 * send_token writes the token length (as a network long) and then the
 * token data to the file descriptor s. It returns 0 on success, and
 * -1 if an error occurs or if it could not write all the data.
 */
int send_token(s, tok)
  int s;
  gss_buffer_t tok;
{
  int len, ret;
  len = htonl(tok->length);
  ret = write_all(s, (char *) &len, 4);
  if (ret < 0) {
    perror("sending token length");
    return -1;
  } else if (ret != 4) {
    if (display_file)
      fprintf(display_file,
           "sending token length: %d of %d bytes written
", ret, 4);
    return -1;
  }
  ret = write_all(s, tok->value, tok->length);
  if (ret < 0) {
    perror("sending token data");
    return -1;
  } else if (ret != tok->length) {
    if (display_file)
      fprintf(display_file,
           "sending token data: %d of %d bytes written
", ret, tok->length);
    return -1;
  }
  return 0;
}
EXAMPLE A–3  Code Listings for Miscellaneous GSS-API Functions  (Continued)

    return -1;
  }
  return 0;
}

/*
 * Function: recv_token
 * Purpose: Reads a token from a file descriptor.
 * Arguments:
 *   s       (r) an open file descriptor
 *   tok     (w) the read token
 * Returns: 0 on success, -1 on failure
 * Effects:
 * recv_token reads the token length (as a network long), allocates
 * memory to hold the data, and then reads the token data from the
 * file descriptor s. It blocks to read the length and data, if
 * necessary. On a successful return, the token should be freed with
 * gss_release_buffer. It returns 0 on success, and -1 if an error
 * occurs or if it could not read all the data.
 */
int recv_token(s, tok)
    int s;
    gss_buffer_t tok;
{
    int ret;
    ret = read_all(s, (char *) &tok->length, 4);
    if (ret < 0) {
        perror("reading token length");
        return -1;
    } else if (ret != 4) {
        if (display_file)
            fprintf(display_file,
                "reading token length: %d of %d bytes read\n",
                ret, 4);
        return -1;
    }
    tok->length = ntohl(tok->length);
    tok->value = (char *) malloc(tok->length);
    if (tok->value == NULL) {
        if (display_file)
            fprintf(display_file,
                "Out of memory allocating token data\n"
            );
        return -1;
    }
    ret = read_all(s, (char *) tok->value, tok->length);
    if (ret < 0) {
EXAMPLE A–3  Code Listings for Miscellaneous GSS-API Functions  (Continued)

perror("reading token data");
free(tok->value);
return -1;
} else if (ret != tok->length) {
    fprintf(stderr, "sending token data: %d of %d bytes written\n",
            ret, tok->length);
    free(tok->value);
    return -1;
}
return 0;
}

static void display_status_1(m, code, type)
char *m;
OM_uint32 code;
int type;
{
OM_uint32 maj_stat, min_stat;
gss_buffer_desc msg;
OM_uint32 msg_ctx;

msg_ctx = 0;
while (1) {
    maj_stat = gss_display_status(&min_stat, code,
        type, GSS_C_NULL_OID,
        &msg_ctx, &msg);
    if (display_file)
        fprintf(display_file, "GSS-API error %s: %s\n",
                m, (char *)msg.value);
    (void) gss_release_buffer(&min_stat, &msg);
    if (!msg_ctx)
        break;
}
}

/*
 * Function: display_status
 * Purpose: displays GSS-API messages
 * Arguments:
 * msg      a string to be displayed with the message
 * maj_stat  the GSS-API major status code
 * min_stat  the GSS-API minor status code
 * Effects:
 * The GSS-API messages associated with maj_stat and min_stat are
 * displayed on stderr, each preceded by "GSS-API error <msg>: " and
 * followed by a newline.
 */
void display_status(msg, maj_stat, min_stat)
char *msg;
EXAMPLE A-3  Code Listings for Miscellaneous GSS-API Functions  (Continued)

    OM_uint32 maj_stat;
    OM_uint32 min_stat;
{
    display_status_1(msg, maj_stat, GSS_C_GSS_CODE);
    display_status_1(msg, min_stat, GSS_C_MECH_CODE);
}

/*
 * Function: display_ctx_flags
 *
 * Purpose: displays the flags returned by context initiation in
 * a human-readable form
 *
 * Arguments:
 * int ret_flags
 *
 * Effects:
 *
 * Strings corresponding to the context flags are printed on
 * stdout, preceded by 'context flag: ' and followed by a newline
 */

void display_ctx_flags(flags)
OM_uint32 flags;
{
    if (flags & GSS_C_DELEG_FLAG)
        fprintf(display_file, "context flag: GSS_C_DELEG_FLAG\n");
    if (flags & GSS_C_MUTUAL_FLAG)
        fprintf(display_file, "context flag: GSS_C_MUTUAL_FLAG\n");
    if (flags & GSS_C_REPLAY_FLAG)
        fprintf(display_file, "context flag: GSS_C_REPLAY_FLAG\n");
    if (flags & GSS_C_SEQUENCE_FLAG)
        fprintf(display_file, "context flag: GSS_C_SEQUENCE_FLAG\n");
    if (flags & GSS_C_CONF_FLAG)
        fprintf(display_file, "context flag: GSS_C_CONF_FLAG \n");
    if (flags & GSS_C_INTEG_FLAG)
        fprintf(display_file, "context flag: GSS_C_INTEG_FLAG \n");
}

void print_token(tok)
gss_buffer_t tok;
{
    int i;
    unsigned char *p = tok->value;
    if (!display_file)
        return;
    for (i=0; i < tok->length; i++, p++) {
        fprintf(display_file, "%02x ", *p);
        if ((i % 16) == 15) {
            fprintf(display_file, "\n");
        }
    }
    fprintf(display_file, "\n");
    fflush(display_file);
 EXAMPLE A–3  Code Listings for Miscellaneous GSS-API Functions  (Continued)

}
This appendix includes the following sections:

- "GSS-API Functions" on page 215 provides a table of GSS-API functions.
- "GSS-API Status Codes" on page 217 discusses status codes returned by GSS-API functions, and provides a list of those status codes.
- "GSS-API Data Types and Values" on page 221 discusses the various data types used by GSS-API.
- "Implementation-Specific Features in GSS-API" on page 224 covers features that are unique to the Oracle Solaris implementation of GSS-API.
- "Kerberos v5 Status Codes" on page 227 lists the status codes that can be returned by the Kerberos v5 mechanism.

Additional GSS-API definitions can be found in the file gssapi.h.

**GSS-API Functions**

The Oracle Solaris software implements the GSS-API functions. For more information on each function, see its man page. See also "Functions From Previous Versions of GSS-API" on page 217.

- `gss_acquire_credential()`: Assume a global identity by obtaining a GSS-API credential handle for pre-existing credentials.
- `gss_add_credential()`: Construct credentials incrementally.
- `gss_inquire_credential()`: Obtain information about a credential.
- `gss_inquire_credential_by_mech()`: Obtain per-mechanism information about a credential.
- `gss_release_credential()`: Discard a credential handle.
- `gss_init_sec_context()`: Initiate a security context with a peer application.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gss_accept_sec_context()</td>
<td>Accept a security context initiated by a peer application</td>
</tr>
<tr>
<td>gss_delete_sec_context()</td>
<td>Discard a security context</td>
</tr>
<tr>
<td>gss_process_context_token()</td>
<td>Process a token on a security context from a peer application</td>
</tr>
<tr>
<td>gss_context_time()</td>
<td>Determine how long a context is to remain valid</td>
</tr>
<tr>
<td>gss_inquire_context()</td>
<td>Obtain information about a security context</td>
</tr>
<tr>
<td>gss_wrap_size_limit()</td>
<td>Determine token-size limit for gss_wrap() on a context</td>
</tr>
<tr>
<td>gss_export_sec_context()</td>
<td>Transfer a security context to another process</td>
</tr>
<tr>
<td>gss_import_sec_context()</td>
<td>Import a transferred context</td>
</tr>
<tr>
<td>gss_get_mic()</td>
<td>Calculate a cryptographic message integrity code (MIC) for a message</td>
</tr>
<tr>
<td>gss_verify_mic()</td>
<td>Check a MIC against a message to verify integrity of a received message</td>
</tr>
<tr>
<td>gss_wrap()</td>
<td>Attach a MIC to a message, and optionally encrypt the message content</td>
</tr>
<tr>
<td>gss_unwrap()</td>
<td>Verify a message with attached MIC. Decrypt message content if necessary</td>
</tr>
<tr>
<td>gss_import_name()</td>
<td>Convert a contiguous string name to an internal-form name</td>
</tr>
<tr>
<td>gss_display_name()</td>
<td>Convert internal-form name to text</td>
</tr>
<tr>
<td>gss_compare_name()</td>
<td>Compare two internal-form names</td>
</tr>
<tr>
<td>gss_release_name()</td>
<td>Discard an internal-form name</td>
</tr>
<tr>
<td>gss_inquire_names_for_mech()</td>
<td>List the name types supported by the specified mechanism</td>
</tr>
<tr>
<td>gss_inquire_mechs_for_name()</td>
<td>List mechanisms that support the specified name type</td>
</tr>
<tr>
<td>gss_canonicalize_name()</td>
<td>Convert an internal name to a mechanism name (MN)</td>
</tr>
<tr>
<td>gss_export_name()</td>
<td>Convert an MN to export form</td>
</tr>
<tr>
<td>gss_duplicate_name()</td>
<td>Create a copy of an internal name</td>
</tr>
<tr>
<td>gss_add_oid_set_member()</td>
<td>Add an object identifier to a set</td>
</tr>
<tr>
<td>gss_display_status()</td>
<td>Convert a GSS-API status code to text</td>
</tr>
<tr>
<td>gss_indicate_mechs()</td>
<td>Determine available underlying authentication mechanisms</td>
</tr>
</tbody>
</table>
Functions From Previous Versions of GSS-API

This section explains functions that were included in previous versions of the GSS-API.

Functions for Manipulating OIDs

The Oracle Solaris implementation of GSS-API provides the following functions for convenience and for backward compatibility. However, these functions might not be supported by other implementations of GSS-API.

- `gss_delete_oid()`
- `gss_oid_to_str()`
- `gss_str_to_oid()`

Although a mechanism’s name can be converted from a string to an OID, programmers should use the default GSS-API mechanism if at all possible.

Renamed Functions

The following functions have been supplanted by newer functions. In each case, the new function is the functional equivalent of the older function. Although the old functions are supported, developers should replace these functions with the newer functions whenever possible.

- `gss_sign()` has been replaced with `gss_get_mic()`.
- `gss_verify()` has been replaced with `gss_verify_mic()`.
- `gss_seal()` has been replaced with `gss_wrap()`.
- `gss_unseal()` has been replaced with `gss_unwrap()`.

GSS-API Status Codes

Major status codes are encoded in the `OM_uint32` as shown in the following figure.
If a GSS-API routine returns a GSS status code whose upper 16 bits contain a nonzero value, the call has failed. If the calling error field is nonzero, the application’s call of the routine was erroneous. The calling errors are listed in Table B–1. If the routine error field is nonzero, the routine failed because of a routine-specific error, as listed in Table B–2. The bits in the supplementary information field of the status code can be set whether the upper 16 bits indicate a failure or a success. The meaning of individual bits is listed in Table B–3.

**GSS-API Major Status Code Values**

The following tables list the calling errors that are returned by GSS-API. These errors are specific to a particular language-binding, which is C in this case.

**TABLE B–1  GSS-API Calling Errors**

<table>
<thead>
<tr>
<th>Error</th>
<th>Value in Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_CALL_INACCESSIBLE_READ</td>
<td>1</td>
<td>An input parameter that is required could not be read</td>
</tr>
<tr>
<td>GSS_S_CALL_INACCESSIBLE_WRITE</td>
<td>2</td>
<td>A required output parameter could not be written</td>
</tr>
<tr>
<td>GSS_S_CALL_BAD_STRUCTURE</td>
<td>3</td>
<td>A parameter was malformed</td>
</tr>
</tbody>
</table>

The following table lists the GSS-API routine errors, generic errors that are returned by GSS-API functions.
### TABLE B–2  GSS-API Routine Errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Value in Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_BAD_MECH</td>
<td>1</td>
<td>An unsupported mechanism was requested.</td>
</tr>
<tr>
<td>GSS_S_BAD_NAME</td>
<td>2</td>
<td>An invalid name was supplied.</td>
</tr>
<tr>
<td>GSS_S_BAD_NAMETYPE</td>
<td>3</td>
<td>A supplied name was of an unsupported type.</td>
</tr>
<tr>
<td>GSS_S_BAD_BINDINGS</td>
<td>4</td>
<td>Incorrect channel bindings were supplied.</td>
</tr>
<tr>
<td>GSS_S_BAD_STATUS</td>
<td>5</td>
<td>An invalid status code was supplied.</td>
</tr>
<tr>
<td>GSS_S_BAD_MIC, GSS_S_BAD_SIG</td>
<td>6</td>
<td>A token had an invalid MIC.</td>
</tr>
<tr>
<td>GSS_S_NO_CRED</td>
<td>7</td>
<td>The credentials were unavailable, inaccessible, or not supplied.</td>
</tr>
<tr>
<td>GSS_S_NO_CONTEXT</td>
<td>8</td>
<td>No context has been established.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_TOKEN</td>
<td>9</td>
<td>A token was invalid.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_CREDENTIAL</td>
<td>10</td>
<td>A credential was invalid.</td>
</tr>
<tr>
<td>GSS_S_CREDENTIALS_EXPIRED</td>
<td>11</td>
<td>The referenced credentials have expired.</td>
</tr>
<tr>
<td>GSS_S_CONTEXT_EXPIRED</td>
<td>12</td>
<td>The context has expired.</td>
</tr>
<tr>
<td>GSS_S_FAILURE</td>
<td>13</td>
<td>Miscellaneous failure. The underlying mechanism detected an error for which no specific GSS-API status code is defined. The mechanism-specific status code, that is, the minor-status code, provides more details about the error.</td>
</tr>
<tr>
<td>GSS_S_BAD_QOP</td>
<td>14</td>
<td>The quality-of-protection that was requested could not be provided.</td>
</tr>
<tr>
<td>GSS_S_UNAUTHORIZED</td>
<td>15</td>
<td>The operation is forbidden by local security policy.</td>
</tr>
<tr>
<td>GSS_S_UNAVAILABLE</td>
<td>16</td>
<td>The operation or option is unavailable.</td>
</tr>
<tr>
<td>GSS_S_DUPLICATE_ELEMENT</td>
<td>17</td>
<td>The requested credential element already exists.</td>
</tr>
<tr>
<td>GSS_S_NAME_NOT_MN</td>
<td>18</td>
<td>The provided name was not a mechanism name (MN).</td>
</tr>
</tbody>
</table>

The name GSS_S_COMPLETE, which is a zero value, indicates an absence of any API errors or supplementary information bits.

The following table lists the supplementary information values returned by GSS-API functions.
### Displaying Status Codes

The function `gss_display_status()` translates GSS-API status codes into text format. This format allows the codes to be displayed to a user or put in a text log. `gss_display_status()` only displays one status code at a time, and some functions can return multiple status conditions. Accordingly, `gss_display_status()` should be called as part of a loop. When `gss_display_status()` indicates a non-zero status code, another status code is available for the function to fetch.

#### EXAMPLE B-1 Displaying Status Codes with `gss_display_status()`

```c
OM_uint32 message_context;
OM_uint32 status_code;
OM_uint32 maj_status;
OM_uint32 min_status;
gss_buffer_desc status_string;
...
message_context = 0;
do {
    maj_status = gss_display_status( 
        &min_status,
        status_code,
        GSS_C_GSS_CODE,
        GSS_C_NO_OID,
        &message_context,
        &status_string);
    fprintf(stderr, "%s\n", \
        (int)status_string.length, \
        status_string.data);
    status_code = gss_get_error(status_code, &message_context);
}
```

For more on status codes, see “GSS-API Status Codes” on page 68.
EXAMPLE B–1  Displaying Status Codes with gss_display_status()  (Continued)

(char *)status_string.value);
gss_release_buffer(&min_status, &status_string,);
} while (message_context != 0);

Status Code Macros

The macros, GSS_CALLING_ERROR(), GSS_ROUTINE_ERROR() and GSS_SUPPLEMENTARY_INFO(), take a GSS status code. These macros remove all information except for the relevant field. For example, the GSS_ROUTINE_ERROR() can be applied to a status code to remove the calling errors and supplementary information fields. This operation leaves the routine errors field only. The values delivered by these macros can be directly compared with a GSS_S_xxx symbol of the appropriate type. The macro GSS_ERROR() returns a non-zero value if a status code indicates a calling or routine error, and a zero value otherwise. All macros that are defined by GSS-API evaluate the arguments exactly once.

GSS-API Data Types and Values

This section describes various types of GSS-API data types and values. Some data types, such as gss_cred_id_t or gss_name_t, are opaque to the user. These data types do not need to be discussed. This section explains the following topics:

- "Basic GSS-API Data Types" on page 221 – Shows the definitions of the OM_uint32, gss_buffer_desc, gss_OID_desc, gss_OID_set_desc_struct, and gss_channel_bindings_struct data types.
- "Name Types" on page 222 – Shows the various name formats recognized by the GSS-API for specifying names.
- "Address Types for Channel Bindings" on page 223 – Shows the various values that can be used as the initiator_addrtype and acceptor_addrtype fields of the gss_channel_bindings_t structure.

Basic GSS-API Data Types

This section describes data types that are used by GSS-API.

OM_uint32
The OM_uint32 is a platform-independent 32-bit unsigned integer.
**gss_buffer_desc**

The definition of the `gss_buffer_desc` with the `gss_buffer_t` pointer takes the following form:

```c
typedef struct gss_buffer_desc_struct {
    size_t length;
    void *value;
} gss_buffer_desc, *gss_buffer_t;
```

**gss_OID_desc**

The definition of the `gss_OID_desc` with the `gss_OID` pointer takes the following form:

```c
typedef struct gss_OID_desc_struct {
    OM_uint32 length;
    void*elements;
} gss_OID_desc, *gss_OID;
```

**gss_OID_set_desc**

The definition of the `gss_OID_set_desc` with the `gss_OID_set` pointer takes the following form:

```c
typedef struct gss_OID_set_desc_struct {
    size_t count;
    gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

**gss_channel_bindings_struct**

The definition of the `gss_channel_bindings_struct` structure and the `gss_channel_bindings_t` pointer has the following form:

```c
typedef struct gss_channel_bindings_struct {
    OM_uint32 initiator_addrtype;
    gss_buffer_desc initiator_address;
    OM_uint32 acceptor_addrtype;
    gss_buffer_desc acceptor_address;
    gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

**Name Types**

A name type indicates the format of the associated name. See “Names in GSS-API” on page 60 and “GSS-API OIDs” on page 66 for more on names and name types. The GSS-API supports the `gss_OID` name types in the following table.

**GSS_C_NO_NAME**

The symbolic name `GSS_C_NO_NAME` is recommended as a parameter value to indicate that no value is supplied in the transfer of names.
GSS_C_NO_OID
This value corresponds to a null input value instead of an actual object identifier. Where specified, the value indicates interpretation of an associated name that is based on a mechanism-specific default printable syntax.

GSS_C_NT_ANONYMOUS
A means to identify anonymous names. This value can be compared with to determine in a mechanism-independent fashion whether a name refers to an anonymous principal.

GSS_C_NT_EXPORT_NAME
A name that has been exported with the gss_export_name() function.

GSS_C_NT_HOSTBASED_SERVICE
Used to represent services that are associated with host computers. This name form is constructed using two elements, service and hostname, as follows: service@hostname.

GSS_C_NT_MACHINE_UID_NAME
Used to indicate a numeric user identifier corresponding to a user on a local system. The interpretation of this value is OS-specific. The gss_import_name() function resolves this UID into a user name, which is then treated as the User Name Form.

GSS_C_NT_STRING_STRING_UID_NAME
Used to indicate a string of digits that represents the numeric user identifier of a user on a local system. The interpretation of this value is OS-specific. This name type is similar to the Machine UID Form, except that the buffer contains a string that represents the user ID.

GSS_C_NT_USER_NAME
A named user on a local system. The interpretation of this value is OS-specific. The value takes the form: username.

Address Types for Channel Bindings
The following table shows the possible values for the initiator_addrtype and acceptor_addrtype fields of the gss_channel_bindings_struct structure. These fields indicate the format that a name can take, for example, ARPAnet IMP address or AppleTalk address. Channel bindings are discussed in “Using Channel Bindings in GSS-API” on page 79.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value (Decimal)</th>
<th>Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_C_AF_UNSPEC</td>
<td>0</td>
<td>Unspecified address type</td>
</tr>
<tr>
<td>GSS_C_AF_LOCAL</td>
<td>1</td>
<td>Host-local</td>
</tr>
<tr>
<td>GSS_C_AF_INET</td>
<td>2</td>
<td>Internet address type, for example, IP</td>
</tr>
<tr>
<td>GSS_C_AF_IMPLINK</td>
<td>3</td>
<td>ARPAnet IMP</td>
</tr>
</tbody>
</table>
TABLE B-4  Channel Binding Address Types  (Continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Value (Decimal)</th>
<th>Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_C_AF_PUP</td>
<td>4</td>
<td>pup protocols, for example, BSP</td>
</tr>
<tr>
<td>GSS_C_AF_CHAOS</td>
<td>5</td>
<td>MIT CHAOS protocol</td>
</tr>
<tr>
<td>GSS_C_AF_NS</td>
<td>6</td>
<td>XEROX NS</td>
</tr>
<tr>
<td>GSS_C_AF_NBS</td>
<td>7</td>
<td>nbs</td>
</tr>
<tr>
<td>GSS_C_AF_ECMA</td>
<td>8</td>
<td>ECMA</td>
</tr>
<tr>
<td>GSS_C_AF_DATAKIT</td>
<td>9</td>
<td>Datakit protocols</td>
</tr>
<tr>
<td>GSS_C_AF_CCITT</td>
<td>10</td>
<td>CCITT</td>
</tr>
<tr>
<td>GSS_C_AF_SNA</td>
<td>11</td>
<td>IBM SNA</td>
</tr>
<tr>
<td>GSS_C_AF_DECnet</td>
<td>12</td>
<td>DECnet</td>
</tr>
<tr>
<td>GSS_C_AF_DLI</td>
<td>13</td>
<td>Direct data link interface</td>
</tr>
<tr>
<td>GSS_C_AF_LAT</td>
<td>14</td>
<td>LAT</td>
</tr>
<tr>
<td>GSS_C_AF_HYLINK</td>
<td>15</td>
<td>NSC Hyperchannel</td>
</tr>
<tr>
<td>GSS_C_AF_APPLETALK</td>
<td>16</td>
<td>AppleTalk</td>
</tr>
<tr>
<td>GSS_C_AF_BSC</td>
<td>17</td>
<td>BISYNC</td>
</tr>
<tr>
<td>GSS_C_AF_DSS</td>
<td>18</td>
<td>Distributed system services</td>
</tr>
<tr>
<td>GSS_C_AF_OSI</td>
<td>19</td>
<td>OSI TP4</td>
</tr>
<tr>
<td>GSS_C_AF_X25</td>
<td>21</td>
<td>X.25</td>
</tr>
<tr>
<td>GSS_C_AF_NULLADDR</td>
<td>255</td>
<td>No address specified</td>
</tr>
</tbody>
</table>

Implementation-Specific Features in GSS-API

Some aspects of the GSS-API can differ between implementations of the API. In most cases, differences in implementations have only minimal effect on programs. In all cases, developers can maximize portability by not relying on any behavior that is specific to a given implementation, including the Oracle Solaris implementation.

Oracle Solaris-Specific Functions

The Oracle implementation does not have customized GSS-API functions.
Human-Readable Name Syntax

Implementations of GSS-API can differ in the printable syntax that corresponds to names. For portability, applications should not compare names that use human-readable, that is, printable, forms. Instead, such applications should use `gss_compare_name()` to determine whether an internal-format name matches any other name.

The Oracle Solaris implementation of `gss_display_name()` displays names as follows. If the `input_name` argument denotes a user principal, the `gss_display_name()` returns `user_principal@realm` as the `output_name_buffer` and the `gss_OID` value as the `output_name_type`. If Kerberos v5 is the underlying mechanism, `gss_OID` is `1.2.840.11354.1.2.2`.

If `gss_display_name()` receives a name that was created by `gss_import_name()` with the `GSS_C_NO_OID` name type, `gss_display_name()` returns `GSS_C_NO_OID` in the `output_name_type` parameter.

Format of Anonymous Names

The `gss_display_name()` function outputs the string `<anonymous>` to indicate an anonymous GSS-API principal. The name type OID associated with this name is `GSS_C_NT_ANONYMOUS`. No other valid printable names supported by the Oracle Solaris implementation should be surrounded by angle brackets (`<>`).

Implementations of Selected Data Types

The following data types have been implemented as pointers, although some implementations might specify these types as arithmetic types: `gss_cred_t`, `gss_ctx_id_t`, and `gss_name_t`.

Deletion of Contexts and Stored Data

When context establishment fails, the Oracle Solaris implementation does not automatically delete partially built contexts. Applications should therefore handle this event by deleting the contexts with `gss_delete_sec_context()`.

This implementation automatically releases stored data, such as internal names, through memory management. However, applications should still call appropriate functions, such as `gss_release_name()`, when data elements are no longer needed.

Protection of Channel-Binding Information

Support for channel bindings varies by mechanism. Both the Diffie-Hellman mechanism and the Kerberos v5 mechanism support channel bindings.
Developers should assume that channel bindings data do not have confidentiality protection. Although the Kerberos v5 mechanism provides this protection, confidentiality for channel-bindings data is not available with the Diffie-Hellman mechanism.

**Context Exportation and Interprocess Tokens**

The Oracle Solaris implementation detects and rejects attempted multiple imports of the same context.

**Types of Credentials Supported**

The Oracle Solaris implementation of the GSS-API supports the acquisition of GSS_C_INITIATE, GSS_C_ACCEPT, and GSS_C_BOTH credentials through `gss_acquire_cred()`.

**Credential Expiration**

The Oracle Solaris implementation of the GSS-API supports credential expiration. Therefore, programmers can use parameters that relate to credential lifetime in functions such as `gss_acquire_cred()` and `gss_add_cred()`.

**Context Expiration**

The Oracle Solaris implementation of the GSS-API supports context expiration. Therefore, programmers can use parameters that relate to context lifetime in functions such as `gss_init_sec_context()` and `gss_inquire_context()`.

**Wrap Size Limits and QOP Values**

The Oracle Solaris implementation of the GSS-API, as opposed to any underlying mechanism, does not impose a maximum size for messages to be processed by `gss_wrap()`. Applications can determine the maximum message size with `gss_wrap_size_limit()`.

The Oracle Solaris implementation of the GSS-API detects invalid QOP values when `gss_wrap_size_limit()` is called.

**Use of minor_status Parameter**

In the Oracle Solaris implementation of the GSS-API, functions return only mechanism-specific information in the `minor_status` parameter. Other implementations might include implementation-specific return values as part of the returned minor-status code.
Kerberos v5 Status Codes

Each GSS-API function returns two status codes: a major status code and a minor status code. Major status codes relate to the behavior of GSS-API. For example, if an application attempts to transmit a message after a security context has expired, GSS-API returns a major status code of GSS_S_CONTEXT_EXPIRED. Major status codes are listed in “GSS-API Status Codes” on page 217.

Minor status codes are returned by the underlying security mechanisms supported by a given implementation of GSS-API. Every GSS-API function takes as the first argument a minor_status or minor_stat parameter. An application can examine this parameter when the function returns, successfully or not, to see the status that is returned by the underlying mechanism.

The following tables list the status messages that can be returned by Kerberos v5 in the minor_status argument. For more on GSS-API status codes, see “GSS-API Status Codes” on page 68.

Messages Returned in Kerberos v5 for Status Code 1

The following table lists the minor status messages that are returned in Kerberos v5 for status code 1.

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5KDC_ERR_NONE</td>
<td>-1765328384L</td>
<td>No error</td>
</tr>
<tr>
<td>KRB5KDC_ERR_NAME_EXP</td>
<td>-1765328383L</td>
<td>Client’s entry in database has expired</td>
</tr>
<tr>
<td>KRB5KDC_ERR_SERVICE_EXP</td>
<td>-1765328382L</td>
<td>Server’s entry in database has expired</td>
</tr>
<tr>
<td>KRB5KDC_ERR_BAD_PVNO</td>
<td>-1765328381L</td>
<td>Requested protocol version not supported</td>
</tr>
<tr>
<td>KRB5KDC_ERR_C_OLD_MAST_KVNO</td>
<td>-1765328380L</td>
<td>Client’s key is encrypted in an old master key</td>
</tr>
<tr>
<td>KRB5KDC_ERR_S_OLD_MAST_KVNO</td>
<td>-1765328379L</td>
<td>Server’s key is encrypted in an old master key</td>
</tr>
<tr>
<td>KRB5KDC_ERR_C_PRINCIPAL_UNKNOWN</td>
<td>-1765328378L</td>
<td>Client not found in Kerberos database</td>
</tr>
<tr>
<td>KRB5KDC_ERR_S_PRINCIPAL_UNKNOWN</td>
<td>-1765328377L</td>
<td>Server not found in Kerberos database</td>
</tr>
</tbody>
</table>
### Kerberos v5 Status Codes

**TABLE B-5  Kerberos v5 Status Codes 1  (Continued)**

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5KDC_ERR_PRINCIPAL_NOT_UNIQUE</td>
<td>-1765328376L</td>
<td>Principal has multiple entries in Kerberos database</td>
</tr>
<tr>
<td>KRB5KDC_ERR_NULL_KEY</td>
<td>-1765328375L</td>
<td>Client or server has a null key</td>
</tr>
<tr>
<td>KRB5KDC_ERR_CANNOT_POSTDATE</td>
<td>-1765328374L</td>
<td>Ticket is ineligible for postdating</td>
</tr>
<tr>
<td>KRB5KDC_ERR_NEVER_VALID</td>
<td>-1765328373L</td>
<td>Requested effective lifetime is negative or too short</td>
</tr>
<tr>
<td>KRB5KDC_ERR_POLICY</td>
<td>-1765328372L</td>
<td>KDC policy rejects request</td>
</tr>
<tr>
<td>KRB5KDC_ERR_BADOPTION</td>
<td>-1765328371L</td>
<td>KDC can’t fulfill requested option</td>
</tr>
<tr>
<td>KRB5KDC_ERR_ETYPE_NOSUPP</td>
<td>-1765328370L</td>
<td>KDC has no support for encryption type</td>
</tr>
<tr>
<td>KRB5KDC_ERR_SUMTYPE_NOSUPP</td>
<td>-1765328369L</td>
<td>KDC has no support for checksum type</td>
</tr>
<tr>
<td>KRB5KDC_ERR_PADATA_TYPE_NOSUPP</td>
<td>-1765328368L</td>
<td>KDC has no support for padata type</td>
</tr>
<tr>
<td>KRB5KDC_ERR_TRTYPE_NOSUPP</td>
<td>-1765328367L</td>
<td>KDC has no support for transited type</td>
</tr>
<tr>
<td>KRB5KDC_ERR_CLIENT_REVOKED</td>
<td>-1765328366L</td>
<td>Client’s credentials have been revoked</td>
</tr>
<tr>
<td>KRB5KDC_ERR_SERVICE_REVOKED</td>
<td>-1765328365L</td>
<td>Credentials for server have been revoked</td>
</tr>
</tbody>
</table>

**Messages Returned in Kerberos v5 for Status Code 2**

The following table lists the minor status messages that are returned in Kerberos v5 for status code 2.

**TABLE B-6  Kerberos v5 Status Codes 2**

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5KDC_ERR_TGT_REVOKED</td>
<td>-1765328364L</td>
<td>TGT has been revoked</td>
</tr>
<tr>
<td>KRB5KDC_ERR_CLIENT_NOTYET</td>
<td>-1765328363L</td>
<td>Client not yet valid, try again later</td>
</tr>
<tr>
<td>Minor Status</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>KB5KDC_ERR_SERVICE_NOTYET</td>
<td>-1765328362L</td>
<td>Server not yet valid, try again later</td>
</tr>
<tr>
<td>KB5KDC_ERR_KEY_EXP</td>
<td>-1765328361L</td>
<td>Password has expired</td>
</tr>
<tr>
<td>KB5KDC_ERR_PREAUTH_FAILED</td>
<td>-1765328360L</td>
<td>Preauthentication failed</td>
</tr>
<tr>
<td>KB5KDC_ERR_PREAUTH_REQUIRED</td>
<td>-1765328359L</td>
<td>Additional preauthentication required</td>
</tr>
<tr>
<td>KB5KDC_ERR_SERVER_NOMATCH</td>
<td>-1765328358L</td>
<td>Requested server and ticket don’t match</td>
</tr>
<tr>
<td>KB5PLACEHOLD_27 through KB5PLACEHOLD_30</td>
<td>-1765328357L through -1765328354L</td>
<td>KB5 error codes 27 through 30 (reserved)</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BAD_INTEGRITY</td>
<td>-1765328353L</td>
<td>Decrypt integrity check failed</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_TKT_EXPIRED</td>
<td>-1765328352L</td>
<td>Ticket expired</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_TKT_NYV</td>
<td>-1765328351L</td>
<td>Ticket not yet valid</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_REPEAT</td>
<td>-1765328350L</td>
<td>Request is a replay</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_NOT_US</td>
<td>-1765328349L</td>
<td>The ticket isn’t for us</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BADMATCH</td>
<td>-1765328348L</td>
<td>Ticket/authenticator do not match</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_SKEW</td>
<td>-1765328347L</td>
<td>Clock skew too great</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BADADDR</td>
<td>-1765328346L</td>
<td>Incorrect net address</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BADVERSION</td>
<td>-1765328345L</td>
<td>Protocol version mismatch</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_MSG_TYPE</td>
<td>-1765328344L</td>
<td>Invalid message type</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_MODIFIED</td>
<td>-1765328343L</td>
<td>Message stream modified</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BADORDER</td>
<td>-1765328342L</td>
<td>Message out of order</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_ILL_CR_TKT</td>
<td>-1765328341L</td>
<td>Illegal cross-realm ticket</td>
</tr>
<tr>
<td>KB5KRB_AP_ERR_BADKEYVER</td>
<td>-1765328340L</td>
<td>Key version is not available</td>
</tr>
</tbody>
</table>

**Messages Returned in Kerberos v5 for Status Code 3**

The following table lists the minor status messages that are returned in Kerberos v5 for status code 3.
# Kerberos v5 Status Codes

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5KRB_AP_ERR_NOKEY</td>
<td>-1765328339L</td>
<td>Service key not available</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_MUT_FAIL</td>
<td>-1765328338L</td>
<td>Mutual authentication failed</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_BADIRECTION</td>
<td>-1765328337L</td>
<td>Incorrect message direction</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_METHOD</td>
<td>-1765328336L</td>
<td>Alternative authentication method required</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_BADSEQ</td>
<td>-1765328335L</td>
<td>Incorrect sequence number in message</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_INAPP_CKSUM</td>
<td>-1765328334L</td>
<td>Inappropriate type of checksum in message</td>
</tr>
<tr>
<td>KRB5PLACEHOLD_51 through KRB5PLACEHOLD_59</td>
<td>-1765328333L through -1765328325L</td>
<td>Kerberos error codes 51 through 59 (reserved)</td>
</tr>
<tr>
<td>KRB5KRB_ERR_GENERIC</td>
<td>-1765328324L</td>
<td>Generic error</td>
</tr>
<tr>
<td>KRB5KRB_ERR_FIELD_TOOLONG</td>
<td>-1765328323L</td>
<td>Field is too long for this implementation</td>
</tr>
<tr>
<td>KRB5PLACEHOLD_62 through KRB5PLACEHOLD_127</td>
<td>-1765328322L through -1765328325L</td>
<td>Kerberos error codes 62 through 127 (reserved)</td>
</tr>
<tr>
<td>value not returned</td>
<td>-1765328256L</td>
<td>For internal use only</td>
</tr>
<tr>
<td>KRB5_LIBOS_BADLOCKFLAG</td>
<td>-1765328255L</td>
<td>Invalid flag for file lock mode</td>
</tr>
<tr>
<td>KRB5_LIBOS_CANTREADPWD</td>
<td>-1765328254L</td>
<td>Cannot read password</td>
</tr>
<tr>
<td>KRB5_LIBOS_BADPWDMATCH</td>
<td>-1765328253L</td>
<td>Password mismatch</td>
</tr>
<tr>
<td>KRB5_LIBOS_PWDINTR</td>
<td>-1765328252L</td>
<td>Password read interrupted</td>
</tr>
<tr>
<td>KRB5_PARSE_ILLCHAR</td>
<td>-1765328251L</td>
<td>Illegal character in component name</td>
</tr>
<tr>
<td>KRB5_PARSE_MALFORMED</td>
<td>-1765328250L</td>
<td>Malformed representation of principal</td>
</tr>
<tr>
<td>KRB5_CONFIG_CANTOPEN</td>
<td>-1765328249L</td>
<td>Can't open/find Kerberos /etc/krb5/krb5 configuration file</td>
</tr>
<tr>
<td>KRB5_CONFIG_BADFORMAT</td>
<td>-1765328248L</td>
<td>Improper format of Kerberos /etc/krb5/krb5 configuration file</td>
</tr>
</tbody>
</table>
### Messages Returned in Kerberos v5 for Status Code 4

The following table lists the minor status messages that are returned in Kerberos v5 for status code 4.

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_CC_UNKNOWN_TYPE</td>
<td>-1765328244L</td>
<td>Unknown credential cache type</td>
</tr>
<tr>
<td>KRB5_CC_NOTFOUND</td>
<td>-1765328243L</td>
<td>No matching credential has been found</td>
</tr>
<tr>
<td>KRB5_CC_END</td>
<td>-1765328242L</td>
<td>End of credential cache reached</td>
</tr>
<tr>
<td>KRB5_NO_TKT_SUPPLIED</td>
<td>-1765328241L</td>
<td>Request did not supply a ticket</td>
</tr>
<tr>
<td>KRB5_KRB_AP_WRONG_PRINC</td>
<td>-1765328240L</td>
<td>Wrong principal in request</td>
</tr>
<tr>
<td>KRB5_KRB_AP_ERR_TKT_INVALID</td>
<td>-1765328239L</td>
<td>Ticket has invalid flag set</td>
</tr>
<tr>
<td>KRB5_PRINC_NOMATCH</td>
<td>-1765328238L</td>
<td>Requested principal and ticket don’t match</td>
</tr>
<tr>
<td>KRB5_KDCREP_MODIFIED</td>
<td>-1765328237L</td>
<td>KDC reply did not match expectations</td>
</tr>
<tr>
<td>KRB5_KDCREP_SKEW</td>
<td>-1765328236L</td>
<td>Clock skew too great in KDC reply</td>
</tr>
<tr>
<td>KRB5_IN_TKT_REALM_MISMATCH</td>
<td>-1765328235L</td>
<td>Client/server realm mismatch in initial ticket request</td>
</tr>
<tr>
<td>KRB5_PROG_ETYPE_NOSUPP</td>
<td>-1765328234L</td>
<td>Program lacks support for encryption type</td>
</tr>
</tbody>
</table>
### Kerberos v5 Status Codes

#### TABLE B–8  Kerberos v5 Status Codes 4

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_PROG_KEYTYPE_NOSUPP</td>
<td>-1765328233L</td>
<td>Program lacks support for key type</td>
</tr>
<tr>
<td>KRB5_WRONG_ETYPE</td>
<td>-1765328232L</td>
<td>Requested encryption type not used in message</td>
</tr>
<tr>
<td>KRB5_PROG_SUMTYPE_NOSUPP</td>
<td>-1765328231L</td>
<td>Program lacks support for checksum type</td>
</tr>
<tr>
<td>KRB5_REALM_UNKNOWN</td>
<td>-1765328230L</td>
<td>Cannot find KDC for requested realm</td>
</tr>
<tr>
<td>KRB5_SERVICE_UNKNOWN</td>
<td>-1765328229L</td>
<td>Kerberos service unknown</td>
</tr>
<tr>
<td>KRB5_KDC_UNREACH</td>
<td>-1765328228L</td>
<td>Cannot contact any KDC for requested realm</td>
</tr>
<tr>
<td>KRB5_NO_LOCALNAME</td>
<td>-1765328227L</td>
<td>No local name found for principal name</td>
</tr>
<tr>
<td>KRB5_MUTUAL_FAILED</td>
<td>-1765328226L</td>
<td>Mutual authentication failed</td>
</tr>
<tr>
<td>KRB5_RC_TYPE_EXISTS</td>
<td>-1765328225L</td>
<td>Replay cache type is already registered</td>
</tr>
<tr>
<td>KRB5_RC_MALLOC</td>
<td>-1765328224L</td>
<td>No more memory to allocate in replay cache code</td>
</tr>
<tr>
<td>KRB5_RC_TYPE_NOTFOUND</td>
<td>-1765328223L</td>
<td>Replay cache type is unknown</td>
</tr>
</tbody>
</table>

---

### Messages Returned in Kerberos v5 for Status Code 5

The following table lists the minor status messages that are returned in Kerberos v5 for status code 5.

#### TABLE B–9  Kerberos v5 Status Codes 5

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_RC_UNKNOWN</td>
<td>-1765328222L</td>
<td>Generic unknown RC error</td>
</tr>
<tr>
<td>KRB5_RC_REPLAY</td>
<td>-1765328221L</td>
<td>Message is a replay</td>
</tr>
<tr>
<td>KRB5_RC_IO</td>
<td>-1765328220L</td>
<td>Replay I/O operation failed</td>
</tr>
<tr>
<td>KRB5_RC_NOIO</td>
<td>-1765328219L</td>
<td>Replay cache type does not support non-volatile storage</td>
</tr>
<tr>
<td>Minor Status</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KRB5_RC_PARSE</td>
<td>-1765328218L</td>
<td>Replay cache name parse and format error</td>
</tr>
<tr>
<td>KRB5_RC_IO_EOF</td>
<td>-1765328217L</td>
<td>End-of-file on replay cache I/O</td>
</tr>
<tr>
<td>KRB5_RC_IO_MALLOC</td>
<td>-1765328216L</td>
<td>No more memory to allocate in replay cache I/O code</td>
</tr>
<tr>
<td>KRB5_RC_IO_PERM</td>
<td>-1765328215L</td>
<td>Permission denied in replay cache code</td>
</tr>
<tr>
<td>KRB5_RC_IO_IO</td>
<td>-1765328214L</td>
<td>I/O error in replay cache i/o code</td>
</tr>
<tr>
<td>KRB5_RC_IO_UNKNOWN</td>
<td>-1765328213L</td>
<td>Generic unknown RC/I/O error</td>
</tr>
<tr>
<td>KRB5_RC_IO_SPACE</td>
<td>-1765328212L</td>
<td>Insufficient system space to store replay information</td>
</tr>
<tr>
<td>KRB5_TRANS_CANTOPEN</td>
<td>-1765328211L</td>
<td>Can’t open/find realm translation file</td>
</tr>
<tr>
<td>KRB5_TRANS_BADFORMAT</td>
<td>-1765328210L</td>
<td>Improper format of realm translation file</td>
</tr>
<tr>
<td>KRB5_LNAME_CANTOPEN</td>
<td>-1765328209L</td>
<td>Can’t open or find lname translation database</td>
</tr>
<tr>
<td>KRB5_LNAME_NOTRANS</td>
<td>-1765328208L</td>
<td>No translation is available for requested principal</td>
</tr>
<tr>
<td>KRB5_LNAME_BADFORMAT</td>
<td>-1765328207L</td>
<td>Improper format of translation database entry</td>
</tr>
<tr>
<td>KRB5_CRYPTO_INTERNAL</td>
<td>-1765328206L</td>
<td>Cryptosystem internal error</td>
</tr>
<tr>
<td>KRB5_KT_BADNAME</td>
<td>-1765328205L</td>
<td>Key table name malformed</td>
</tr>
<tr>
<td>KRB5_KT_UNKNOWN_TYPE</td>
<td>-1765328204L</td>
<td>Unknown Key table type</td>
</tr>
<tr>
<td>KRB5_KT_NOTFOUND</td>
<td>-1765328203L</td>
<td>Key table entry not found</td>
</tr>
<tr>
<td>KRB5_KT_END</td>
<td>-1765328202L</td>
<td>End of key table reached</td>
</tr>
<tr>
<td>KRB5_KT_NOWRITE</td>
<td>-1765328201L</td>
<td>Cannot write to specified key table</td>
</tr>
</tbody>
</table>
The following table lists the minor status messages that are returned in Kerberos v5 for status code 6.

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_KT_IOERR</td>
<td>-1765328200L</td>
<td>Error writing to key table</td>
</tr>
<tr>
<td>KRB5_NO_TKT_IN_RLM</td>
<td>-1765328199L</td>
<td>Cannot find ticket for requested realm</td>
</tr>
<tr>
<td>KRB5DES_BAD_KEYPAR</td>
<td>-1765328198L</td>
<td>DES key has bad parity</td>
</tr>
<tr>
<td>KRB5DES_WEAK_KEY</td>
<td>-1765328197L</td>
<td>DES key is a weak key</td>
</tr>
<tr>
<td>KRB5_BAD_ENCTYPE</td>
<td>-1765328196L</td>
<td>Bad encryption type</td>
</tr>
<tr>
<td>KRB5_BAD_KEYSIZE</td>
<td>-1765328195L</td>
<td>Key size is incompatible with encryption type</td>
</tr>
<tr>
<td>KRB5_BAD_MSIZE</td>
<td>-1765328194L</td>
<td>Message size is incompatible with encryption type</td>
</tr>
<tr>
<td>KRB5_CC_TYPE_EXISTS</td>
<td>-1765328193L</td>
<td>Credentials cache type is already registered</td>
</tr>
<tr>
<td>KRB5_KT_TYPE_EXISTS</td>
<td>-1765328192L</td>
<td>Key type is already registered</td>
</tr>
<tr>
<td>KRB5_CC_IO</td>
<td>-1765328191L</td>
<td>Credentials cache I/O operation failed</td>
</tr>
<tr>
<td>KRB5_FCC_PERM</td>
<td>-1765328190L</td>
<td>Credentials cache file permissions incorrect</td>
</tr>
<tr>
<td>KRB5_FCC_NOFILE</td>
<td>-1765328189L</td>
<td>No credentials cache file found</td>
</tr>
<tr>
<td>KRB5_FCC_INTERNAL</td>
<td>-1765328188L</td>
<td>Internal file credentials cache error</td>
</tr>
<tr>
<td>KRB5_CC_WRITE</td>
<td>-1765328187L</td>
<td>Error writing to credentials cache file</td>
</tr>
<tr>
<td>KRB5_CC_NOMEM</td>
<td>-1765328186L</td>
<td>No more memory to allocate in credentials cache code</td>
</tr>
<tr>
<td>KRB5_CC_FORMAT</td>
<td>-1765328185L</td>
<td>Bad format in credentials cache</td>
</tr>
</tbody>
</table>
### TABLE B–10  Kerberos v5 Status Codes 6  (Continued)

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_INVALID_FLAGS</td>
<td>-1765328184L</td>
<td>Invalid KDC option combination, which is an internal library error</td>
</tr>
<tr>
<td>KRB5_NO_2ND_TKT</td>
<td>-1765328183L</td>
<td>Request missing second ticket</td>
</tr>
<tr>
<td>KRB5_NOCREDS_SUPPLIED</td>
<td>-1765328182L</td>
<td>No credentials supplied to library routine</td>
</tr>
<tr>
<td>KRB5_SENDAUTH_BADAUTHVERS</td>
<td>-1765328181L</td>
<td>Bad sendauth version was sent</td>
</tr>
<tr>
<td>KRB5_SENDAUTH_BADAPPLVERS</td>
<td>-1765328180L</td>
<td>Bad application version was sent by sendauth</td>
</tr>
<tr>
<td>KRB5_SENDAUTH_BADRESPONSE</td>
<td>-1765328179L</td>
<td>Bad response during sendauth exchange</td>
</tr>
<tr>
<td>KRB5_SENDAUTH_REJECTED</td>
<td>-1765328178L</td>
<td>Server rejected authentication during sendauth exchange</td>
</tr>
</tbody>
</table>

### Messages Returned in Kerberos v5 for Status Code 7

The following table lists the minor status messages that are returned in Kerberos v5 for status code 7.

### TABLE B–11  Kerberos v5 Status Codes 7

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_PREAUTH_BAD_TYPE</td>
<td>-1765328177L</td>
<td>Unsupported preauthentication type</td>
</tr>
<tr>
<td>KRB5_PREAUTH_NO_KEY</td>
<td>-1765328176L</td>
<td>Required preauthentication key not supplied</td>
</tr>
<tr>
<td>KRB5_PREAUTH_FAILED</td>
<td>-1765328175L</td>
<td>Generic preauthentication failure</td>
</tr>
<tr>
<td>KRB5_RCACHE_BADVNO</td>
<td>-1765328174L</td>
<td>Unsupported format version number for replay cache</td>
</tr>
<tr>
<td>KRB5_CCACHE_BADVNO</td>
<td>-1765328173L</td>
<td>Unsupported credentials cache format version number</td>
</tr>
<tr>
<td>Minor Status</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KRB5_KEYTAB_BADVNO</td>
<td>-1765328172L</td>
<td>Unsupported version number for key table format</td>
</tr>
<tr>
<td>KRB5_PROG_ATYPE_NOSUPP</td>
<td>-1765328171L</td>
<td>Program lacks support for address type</td>
</tr>
<tr>
<td>KRB5_RC_REQUIRED</td>
<td>-1765328170L</td>
<td>Message replay detection requires rcache parameter</td>
</tr>
<tr>
<td>KRB5_ERR_BAD_HOSTNAME</td>
<td>-1765328169L</td>
<td>Host name cannot be canonicalized</td>
</tr>
<tr>
<td>KRB5_ERR_HOST_REALM_UNKNOWN</td>
<td>-1765328168L</td>
<td>Cannot determine realm for host</td>
</tr>
<tr>
<td>KRB5_SNAME_UNSUPP_NAMETYPE</td>
<td>-1765328167L</td>
<td>Conversion to service principal is undefined for name type</td>
</tr>
<tr>
<td>KRB5KRB_AP_ERR_V4_REPLY</td>
<td>-1765328166L</td>
<td>Initial Ticket response appears to be Version 4 error</td>
</tr>
<tr>
<td>KRB5_REALM_CANT_RESOLVE</td>
<td>-1765328165L</td>
<td>Cannot resolve KDC for requested realm</td>
</tr>
<tr>
<td>KRB5_TKT_NOT_FORWARDABLE</td>
<td>-1765328164L</td>
<td>The requesting ticket cannot get forwardable tickets</td>
</tr>
<tr>
<td>KRB5_FWD_BAD_PRINCIPAL</td>
<td>-1765328163L</td>
<td>Bad principal name while trying to forward credentials</td>
</tr>
<tr>
<td>KRB5_GET_IN_TKT_LOOP</td>
<td>-1765328162L</td>
<td>Looping detected inside krb5_get_in_tkt</td>
</tr>
<tr>
<td>KRB5_CONFIG_NODEFREALM</td>
<td>-1765328161L</td>
<td>Configuration file /etc/krb5/krb5.conf does not specify default realm</td>
</tr>
<tr>
<td>KRB5_SAM_UNSUPPORTED</td>
<td>-1765328160L</td>
<td>Bad SAM flags in obtain Sam_padata</td>
</tr>
<tr>
<td>KRB5_KT_NAME_TOOLONG</td>
<td>-1765328159L</td>
<td>Keytab name too long</td>
</tr>
<tr>
<td>KRB5_KT_KVNONOTFOUND</td>
<td>-1765328158L</td>
<td>Key version number for principal in key table is incorrect</td>
</tr>
</tbody>
</table>
### Kerberos v5 Status Codes

#### TABLE B–11  Kerberos v5 Status Codes 7  (Continued)

<table>
<thead>
<tr>
<th>Minor Status</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRB5_CONF_NOT_CONFIGURED</td>
<td>-1765328157L</td>
<td>Kerberos /etc/krb5/krb5.conf configuration file not configured</td>
</tr>
<tr>
<td>ERROR_TABLE_BASE_krb5</td>
<td>-1765328384L</td>
<td>default</td>
</tr>
</tbody>
</table>
You should use the default QOP and mechanism provided by the GSS-API if at all possible. See "GSS-API OIDs" on page 66. However, you might have your own reasons for specifying OIDs. This appendix describes how to specify OIDs. The following topics are covered.

- "Files with OID Values” on page 239
- “Constructing Mechanism OIDs” on page 241
- “Specifying a Non-Default Mechanism” on page 243

Files with OID Values

For convenience, the GSS- API does allow mechanisms and QOPs to be displayed in human-readable form. On Solaris systems, two files, /etc/gss/mech and /etc/gss/qop, contain information about available mechanisms and available QOPs. If you do not have access to these files, then you must provide the string literals from some other source. The published Internet standard for that mechanism or QOP should serve that purpose.

/etc/gss/mech File

The /etc/gss/mech file lists the mechanisms that are available. /etc/gss/mech contains the names in both the numerical format and the alphabetic form. /etc/gss/mech presents the information in this format:

- Mechanism name, in ASCII
- Mechanism’s OID
- Shared library for implementing the services that are provided by this mechanism
- Optionally, the kernel module for implementing the service

A sample /etc/gss/mech might look like Example C–1.
EXAMPLE C-1 The /etc/gss/mech File

```
# Copyright 2003 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# This file contains the GSS-API based security mechanism names,
# the associated object identifiers (OID) and a shared library that
# implements the services for the mechanisms under GSS-API.
#
# Mechanism Name       Object Identifier       Shared Library   Kernel Module
# kerberos_v5          1.2.840.113554.1.2.2    mech_krb5.so     kmech_krb5
# spnego                1.3.6.1.5.5.2          mech_spnego.so.1  [msinterop]
# diffie_hellman_640_0  1.3.6.4.1.42.2.26.2.4  dh640-0.so.1     
# diffie_hellman_1024_0 1.3.6.4.1.42.2.26.2.5   dh1024-0.so.1    
```

/etc/gss/qop File

The /etc/gss/qop file stores, for all mechanisms installed, all the QOPs supported by each mechanism, both as an ASCII string and as the corresponding 32-bit integer. A sample /etc/gss/qop might look like the following example.

EXAMPLE C-2 The /etc/gss/qop File

```
# Copyright (c) 2000, by Sun Microsystems, Inc.
# All rights reserved.
#
# This file contains information about the GSS-API based quality of
# protection (QOP), its string name and its value (32-bit integer).
#
# QOP string QOP Value Mechanism Name
# GSS_KRB5_INTEG_C_QOP DES MD5 0     kerberos_v5
# GSS_KRB5_CONF_C_QOP DES     0     kerberos_v5
```

gss_str_to_oid() Function

For backward compatibility with earlier versions of the GSS-API, this implementation of the GSS-API supports the function `gss_str_to_oid()`. `gss_str_to_oid()` converts a string that represents a mechanism or QOP to an OID. The string can be either as a number or a word.
Caution–gss_str_to_oid(), gss_oid_to_str(), and gss_release_oid() are not supported by some implementations of the GSS-API to discourage the use of explicit, non-default mechanisms and QOPs.

The mechanism string can be hard-coded in the application or come from user input. However, not all implementations of the GSS-API support gss_str_to_oid(), so applications should not rely on this function.

The number that represents a mechanism can have two different formats. The first format, {1 2 3 4}, is officially mandated by the GSS-API specifications. The second format, 1.2.3.4, is more widely used but is not an official standard format. gss_str_to_oid() expects the mechanism number in the first format, so you must convert the string if the string is in the second format before calling gss_str_to_oid(). An example of gss_str_to_oid() is shown in Example C–3. If the mechanism is not a valid one, gss_str_to_oid() returns GSS_S_BAD_MECH.

Because gss_str_to_oid() allocates GSS-API data space, the gss_release_oid() function exists to provide the removed allocated OID when you are finished. Like gss_str_to_oid(), gss_release_oid() is not a generally supported function and should not be relied upon in programs that aspire to universal portability.

Constructing Mechanism OIDs

Because gss_str_to_oid() cannot always be used, there are alternative techniques for finding and selecting mechanisms. One way is to construct a mechanism OID manually and then compare that mechanism to a set of available mechanisms. Another way is to get the set of available mechanisms and choose one from the set.

The gss_OID type has the following form:

```c
typedef struct gss_OID_desc {
    OM_uint32 length;
    void *elements;
} gss_OID_desc, *gss_OID;
```

where the elements field of this structure points to the first byte of an octet string containing the ASN.1 BER encoding of the value portion of the normal BER TLV encoding of the gss_OID. The length field contains the number of bytes in this value. For example, the gss_OID value that corresponds to the DASS X.509 authentication mechanism has a length field of 7 and an elements field that points to the following octal values: 53, 14, 2, 207, 163, 7, 5.

One way to construct a mechanism OID is to declare a gss_OID and then initialize the elements manually to represent a given mechanism. As above, the input for the elements values can be hard-coded, obtained from a table, or entered by a user. This method is somewhat more painstaking than using gss_str_to_oid() but achieves the same effect.
This constructed gss_OID can then be compared against a set of available mechanisms that have been returned by the functions gss_indicate_mechs() or gss_inquire_mechs_for_name(). The application can check for the constructed mechanism OID in this set of available mechanisms by using the gss_test_oid_set_member() function. If gss_test_oid_set_member() does not return an error, then the constructed OID can be used as the mechanism for GSS-API transactions.

As an alternative to constructing a preset OID, the application can use gss_indicate_mechs() or gss_inquire_mechs_for_name() to get the gss_OID_set of available mechanisms. A gss_OID_set has the following form:

typedef struct gss_OID_set_desc_struct {
    OM_uint32 length;
    void *elements;
} gss_OID_set_desc, *gss_OID_set;

where each of the elements is a gss_OID that represents a mechanism. The application can then parse each mechanism and display the numerical representation. A user can use this display to choose the mechanism. The application then sets the mechanism to the appropriate member of the gss_OID_set. The application can also compare the desired mechanisms against a list of preferred mechanisms.

**createMechOid() Function**

This function is shown for the sake of completeness. Normally, you should use the default mechanism, which is specified by GSS_C_NULL_OID.

```
EXAMPLEC-3    createMechOid() Function

gss_OID createMechOid(const char *mechStr)
{
    gss_buffer_desc mechDesc;
    gss_OID mechOid;
    OM_uint32 minor;
    if (mechStr == NULL)
        return (GSS_C_NULL_OID);

    mechDesc.length = strlen(mechStr);
    mechDesc.value = (void *) mechStr;
    if (gss_str_to_oid(&minor, &mechDesc, &mechOid) != GSS_S_COMPLETE) {
        fprintf(stderr, "Invalid mechanism oid specified <%s>", mechStr);
        return (GSS_C_NULL_OID);
    }

    return (mechOid);
}
```
Specifying a Non-Default Mechanism

parse_oid() converts the name of a security mechanism on the command line to a compatible OID.

EXAMPLEC-4 parse_oid() Function

static void parse_oid(char *mechanism, gss_OID *oid)
{
    char *mechstr = 0, *cp;
    gss_buffer_desc tok;
    OM_uint32 maj_stat, min_stat;
    if (isdigit(mechanism[0])) {
        mechstr = malloc(strlen(mechanism)+5);
        if (!mechstr) {
            printf("Couldn't allocate mechanism scratch!\n");
            return;
        }
        sprintf(mechstr, "{%s}", mechanism);
        for (cp = mechstr; *cp; cp++)
            if (*cp == '.')
                *cp = ' ';
        tok.value = mechstr;
    } else
        tok.value = mechanism;
    tok.length = strlen(tok.value);
    maj_stat = gss_str_to_oid(&min_stat, &tok, oid);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("str_to_oid", maj_stat, min_stat);
        return;
    }
    if (mechstr)
        free(mechstr);
}
This appendix contains the source code for the example in “SASL Example” on page 135. The appendix includes the following topics:

- “SASL Client Example” on page 245
- “SASL Server Example” on page 253
- “Common Code” on page 261

SASL Client Example

The following code listing is for the sample client in “SASL Example” on page 135.

```c
#pragma ident "@(#)client.c 1.4 03/04/07 SMI"
/* $Id: client.c,v 1.3 2002/09/03 15:11:59 rjs3 Exp $ */
/*
* Copyright (c) 2001 Carnegie Mellon University. All rights reserved.
* 
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
* 
* 3. The name "Carnegie Mellon University" must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For permission or any other legal
*    details, please contact
*    Office of Technology Transfer
*    Carnegie Mellon University
*    5000 Forbes Avenue
*    Pittsburgh, PA 15213-3890
```
SASLClientExample

/* remove \r\n at end of the line */
static void chop(char *s)
{
    char *p;

    assert(s);
    p = s + strlen(s) - 1;
    if (p[0] == '\n') {
        *p-- = '\0';
    } else if (p >= s && p[0] == '\r') {
        *p-- = '\0';
    }
}
static int getrealm(void *context __attribute__((unused)),
    int id,
    const char **availrealms,
    const char **result)
{
    static char buf[1024];

    /* Double-check the ID */
    if (id != SASL_CB_GETREALM) return SASL_BADPARAM;
    if (!result) return SASL_BADPARAM;

    printf("please choose a realm (available:");
    while (*availrealms) {
        printf("%s", *availrealms);
        availrealms++;
    }
    printf("");

    fgets(buf, sizeof buf, stdin);
    chop(buf);
    *result = buf;

    return SASL_OK;
}

static int simple(void *context __attribute__((unused)),
    int id,
    const char **result,
    unsigned *len)
{
    static char buf[1024];

    /* Double-check the connection */
    if (! result)
        return SASL_BADPARAM;

    switch (id) {
    case SASL_CB_USER:
        printf("please enter an authorization id: ");
        break;
    case SASL_CB_AUTHNAME:
        printf("please enter an authentication id: ");
        break;
    default:
        return SASL_BADPARAM;
    }

    fgets(buf, sizeof buf, stdin);
    chop(buf);
    *result = buf;
    if (len) *len = strlen(buf);

    return SASL_OK;
}

#ifndef HAVE_GETPASSPHRASE
static char *
getpassphrase(const char *prompt)
{

return getpass(prompt);
}
#endif /* ! HAVE_GETPASSPHRASE */

static int
getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}

static int getpath(void *context __attribute__((unused)),
    const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    { SASL_CB_GETREALM, &getrealm, NULL },

    /* callbacks we support */
    static int getpath(void *context __attribute__((unused)),
        const char **path)
        return getpass(prompt);
    }
#endif /* ! HAVE_GETPASSPHRASE */

static int getsecret(sasl_conn_t *conn,
    void *context __attribute__((unused)),
    int id,
    sasl_secret_t **psecret)
{
    char *password;
    size_t len;
    static sasl_secret_t *x;
    /* paranoia check */
    if (!conn || !psecret || id != SASL_CB_PASS)
        return SASL_BADPARAM;

    password = getpassphrase("Password: ");
    if (!password)
        return SASL_FAIL;

    len = strlen(password);
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    if (!x) {
        memset(password, 0, len);
        return SASL_NOMEM;
    }
    x->len = len;
    #ifdef _SUN_SDK_
    strcpy((char *)x->data, password);
    #else
    strcpy(x->data, password);
    #endif /* _SUN_SDK_ */
    memset(password, 0, len);
    *psecret = x;
    return SASL_OK;
}
SASL_CB_USER, &simple, NULL
), {
   SASL_CB_AUTNAME, &simple, NULL
}, {
   SASL_CB_PASS, &getsecret, NULL
}, {
   SASL_CB_GETPATH, &getpath, NULL
}, {
   SASL_CB_LIST_END, NULL, NULL
}
);

int getconn(const char *host, const char *port)
{
   struct addrinfo hints, *ai, *r;
   int err, sock = -1;

   memset(&hints, 0, sizeof(hints));
   hints.ai_family = PF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;

   if ((err = getaddrinfo(host, port, &hints, &ai)) != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
      exit(EX_UNAVAILABLE);
   }

   for (r = ai; r; r = r->ai_next) {
      sock = socket(r->ai_family, r->ai_socktype, r->ai_protocol);
      if (sock < 0)
         continue;
      if (connect(sock, r->ai_addr, r->ai_addrlen) >= 0)
         break;
      close(sock);
      sock = -1;
   }

   freeaddrinfo(ai);
   if (sock < 0) {
      perror("connect");
      exit(EX_UNAVAILABLE);
   }

   return sock;
}

char *mech;

int mysasl_negotiate(FILE *in, FILE *out, sasl_conn_t *conn)
{
   char buf[2048];
   const char *data;
   const char *chosenmech;
   #ifdef _SUN_SDK_
   unsigned len;
   #else
   int len;
   #endif /* _SUN_SDK_ */
   int r, c;

/* get the capability list */
dprintf(0, "receiving capability list...");
len = recv_string(in, buf, sizeof buf);
dprintf(0, "%s\n", buf);

if (mech) {
    /* make sure that 'mech' appears in 'buf' */
    if (!strstr(buf, mech)) {
        printf("server doesn't offer mandatory mech '%s'\n", mech);
        return -1;
    }
}
else {
    mech = buf;
}

r = sasl_client_start(conn, mech, NULL, &data, &len, &chosenmech);
if (r != SASL_OK && r != SASL_CONTINUE) {
    saslerr(r, "starting SASL negotiation");
    printf("\n%s\n", sasl_errdetail(conn));
    return -1;
}
dprintf(1, "using mechanism %s\n", chosenmech);

/* we send up to 3 strings; 
   the mechanism chosen, the presence of initial response, 
   and optionally the initial response */ 
send_string(out, chosenmech, strlen(chosenmech));
if(data) {
    send_string(out, "Y", 1);
    send_string(out, data, len);
} else {
    send_string(out, "N", 1);
}

for (;;) {
    c = fgetc(in);
    switch (c) {
    case 'O':
        goto done_ok;
    case 'N':
        goto done_no;
    case 'C': /* continue authentication */
        break;
    default:
        printf("bad protocol from server (%c %x)\n", c, c);
        return -1;
    }
}
len = recv_string(in, buf, sizeof buf);

r = sasl_client_step(conn, buf, len, NULL, &data, &len);
if (r != SASL_OK && r != SASL_CONTINUE) {
    saslerr(r, "performing SASL negotiation");
    printf("\n%s\n", sasl_errdetail(conn));
return -1;
}
if (data) {
    dprintf(2, "sending response length %d...\n", len);
    send_string(out, data, len);
} else {
    dprintf(2, "sending null response...\n");
    send_string(out, ",", 0);
}
}
done_ok:
    printf("successful authentication\n");
    return 0;
done_no:
    printf("authentication failed\n");
    return -1;
}
#endif /* _SUN_SDK_ */
void usage(const char *s)
#else
void usage(void)
#endif /* _SUN_SDK_ */
{
    #ifdef _SUN_SDK_
    fprintf(stderr, "usage: %s [-p port] [-s service] [-m mech] host\n", s);
    #else
    fprintf(stderr, "usage: client [-p port] [-s service] \n"[-m mech] host\n");
    #endif /* _SUN_SDK_ */
exit(EX_USAGE);
}
int main(int argc, char *argv[])
{
    int c;
    char *host = "localhost";
    char *port = "12345";
    char localaddr[NI_MAXHOST + NI_MAXSERV],
    remoteaddr[NI_MAXHOST + NI_MAXSERV];
    char *service = "rcmd";
    char hbuf[NI_MAXHOST], pbuf[NI_MAXSERV];
    int r;
    sasl_conn_t *conn;
    FILE *in, *out;
    int fd;
    int salen;
    struct sockaddr_storage local_ip, remote_ip;
    while ((c = getopt(argc, argv, "p:s:m:")) != EOF) {
        switch(c) {
        case 'p':
            port = optarg;
            break;
        case 's':
            break;
        case 'm':
            break;
        case 'h':
            break;
        case 'o':
            break;
        case 'u':
            break;
        case 'v':
            break;
        default:
            break;
        }
service = optarg;
break;

case 'm':
    mech = optarg;
    break;

default:
    #ifdef _SUN_SDK
        usage(argv[0]);
    #else
        usage();
    #endif /* _SUN_SDK */
    break;
}
}

if (optind > argc - 1) {
    #ifdef _SUN_SDK
        usage(argv[0]);
    #else
        usage();
    #endif /* _SUN_SDK */
    if (optind == argc - 1) {
        host = argv[optind];
    }

    /* initialize the sasl library */
    r = sasl_client_init(callbacks);
    if (r != SASL_OK) saslfail(r, "initializing libsasl");

    /* connect to remote server */
    fd = getconn(host, port);

    /* set ip addresses */
    salen = sizeof(local_ip);
    if (getsockname(fd, (struct sockaddr *)&local_ip, &salen) < 0) {
        perror("getsockname");
    }

    getnameinfo((struct sockaddr *)&local_ip, salen,
                hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),
    #ifdef _SUN_SDK /* SOLARIS doesn't support NI_WITHSCOPEID */
        NI_NUMERICHOST | NI_NUMERICSERV);
    #else
        NI_NUMERICHOST | NI_NUMERICSERV;
    #endif
    snprintf(localaddr, sizeof(localaddr), "%s;%s", hbuf, pbuf);

    salen = sizeof(remote_ip);
    if (getpeername(fd, (struct sockaddr *)&remote_ip, &salen) < 0) {
        perror("getpeername");
    }

    getnameinfo((struct sockaddr *)&remote_ip, salen,
                hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),
    #ifdef _SUN_SDK /* SOLARIS doesn't support NI_WITHSCOPEID */
        NI_NUMERICHOST | NI_NUMERICSERV);
    #else
        NI_NUMERICHOST | NI_NUMERICSERV;
    #endif
    }
else
    NI_NUMERICHOST | NI_WITHSCOPEID | NI_NUMERICSERV);
#endif

snprintf(remoteaddr, sizeof(remoteaddr), "%s;%s", hbuf, pbuf);

/* client new connection */
  r = sasl_client_new(service, host, localaddr, remoteaddr, NULL, 
                    0, &conn);
  if (r != SASL_OK) saslfail(r, "allocating connection state");

/* set external properties here
  sasl_setprop(conn, SASL_SSF_EXTERNAL, &extprops); */

/* set required security properties here
  sasl_setprop(conn, SASL_SEC_PROPS, &secprops); */

  in = fdopen(fd, "r");
  out = fdopen(fd, "w");
  r = mysasl_negotiate(in, out, conn);
  if (r == SASL_OK) {
      /* send/receive data */

      printf("closing connection\n");
      fclose(in);
      fclose(out);
      close(fd);
      sasl_dispose(&conn);
      sasl_done();

      return 0;
  }

SASL Server Example

The following code listing is for the sample server in "SASL Example" on page 135.

#pragma ident "@(#)server.c 1.3 03/04/07 SMI"
/* $Id: server.c,v 1.4 2002/10/07 05:04:05 rjs3 Exp $ */
/* Copyright (c) 2001 Carnegie Mellon University. All rights reserved. *
* Redistribution and use in source and binary forms, with or without *
* modification, are permitted provided that the following conditions *
* are met: *
* 1. Redistributions of source code must retain the above copyright *
*    notice, this list of conditions and the following disclaimer. *
* 2. Redistributions in binary form must reproduce the above copyright *
*    notice, this list of conditions and the following disclaimer in
* 1. Redistributions of any kind must reproduce the following
*    acknowledgment:
*    "This product includes software developed by Computing Services
*     at Carnegie Mellon University (http://www.cmu.edu/computing/)."
*    CARNEGIE MELLON UNIVERSITY DISCLAIMS ALL WARRANTIES WITH REGARD TO
*    THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
*    AND FITNESS, IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE
*    FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
*    WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
*    AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
*    OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <config.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <ctype.h>
#include <errno.h>
#include <string.h>
ifdef HAVE_UNISTD_H
#include <unistd.h>
endif /* HAVE_UNISTD_H */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
ifdef _SUN_SDK_
#include <sysexits.h>
#endif /* _SUN_SDK_ */
#include <sasl.h>
#include "common.h"

 ifdef IPV6_BINDV6ONLY & defined(IN6P_IPV6_V6ONLY)
define IPV6_BINDV6ONLY IN6P_BINDV6ONLY
 endif
 ifndef IPV6_V6ONLY & defined(IPV6_BINDV6ONLY)
define IPV6_V6ONLY IPV6_BINDV6ONLY
 endif

SASLServerExample

Developer's Guide to Oracle Solaris 10 Security • September 2013

254
```c
#ifndef IPV6_BINDV6ONLY
#undef IPV6_V6ONLY
#endif

static int getpath(void * context __attribute__((unused)),
        const char **path)
{
    *path = getenv("SASL_PATH");
    if (*path == NULL)
        *path = PLUGINDIR;
    return SASL_OK;
}

/* callbacks we support */
static sasl_callback_t callbacks[] = {
    {SASL_CB_GETPATH, &getpath, NULL},
    {SASL_CB_LIST_END, NULL, NULL}
};

/* create a socket listening on port 'port' */
/* if af is PF_UNSPEC more than one socket might be returned */
/* the returned list is dynamically allocated, so caller needs to free it */
int *listensock(const char *port, const int af)
{
    struct addrinfo hints, *ai, *r;
    int err, maxs, *sock, *socks;
    const int on = 1;
    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = af;
    hints.ai_socktype = SOCK_STREAM;
    err = getaddrinfo(NULL, port, &hints, &ai);
    if (err) {
        fprintf(stderr, "%s\n", gai_strerror(err));
        exit(EX_USAGE);
    }
    /* Count max number of sockets we can open */
    for (maxs = 0, r = ai; r; r = r->ai_next, maxs++)
        ;
    socks = malloc((maxs + 1) * sizeof(int));
    if (!socks) {
        fprintf(stderr, "can't allocate memory for sockets\n");
        freeaddrinfo(ai);
        exit(EX_OSERR);
    }
    socks[0] = 0; /* num of sockets counter at start of array */
    sock = socks + 1;
    for (r = ai; r; r = r->ai_next) {
        fprintf(stderr, "trying %d, %d, %d\n", r->ai_family, r->ai_socktype,
                r->ai_protocol);
        *sock = socket(r->ai_family, r->ai_socktype, r->ai_protocol);
        sock = *sock + 1;
    }
}
```

if (*sock < 0) {
    perror("socket");
    continue;
}
if (setsockopt(*sock, SOL_SOCKET, SO_REUSEADDR,
    (void *) &on, sizeof(on)) < 0) {
    perror("setsockopt(SO_REUSEADDR)\n");
    close(*sock);
    continue;
}
#endif
if (bind(*sock, r->ai_addr, r->ai_addrlen) < 0) {
    perror("bind\n");
    close(*sock);
    continue;
}
if (listen(*sock, 5) < 0) {
    perror("listen\n");
    close(*sock);
    continue;
}
socks[0]++;
sock++;
}
freeaddrinfo(ai);
if (socks[0] == 0) {
    fprintf(stderr, "Couldn't bind to any socket\n"");
    free(socks);
    exit(EX_OSERR);
}
return socks;
#endif /* _SUN_SDK_ */
#else
    fprintf(stderr,
        "usage: server \[-p port\] \[-s service\] \[-m mech\]\n"
    );
    exit(EX_USAGE);
#endif /* _SUN_SDK_ */
/* Globals are used here, but local variables are preferred */
char *mech;

/* do the sasl negotiation; return -1 if it fails */
int mysasl_negotiate(FILE *in, FILE *out, sasl_conn_t *conn) {
    char buf[8192];
    char chosenmech[128];
    const char *data;
#ifdef _SUN_SDK_
    unsigned len;
#else
    int len;
#endif /* _SUN_SDK_ */
    int r = SASL_FAIL;
    const char *userid;

    /* generate the capability list */
    if (mech) {
        dprintf(2, "forcing use of mechanism %s\n", mech);
        data = strdup(mech);
    } else {
        int count;
        dprintf(1, "generating client mechanism list...\n");
        r = sasl_listmech(conn, NULL, NULL, "", NULL, &data, &len, &count);
        if (r != SASL_OK) saslfail(r, "generating mechanism list");
        dprintf(1, "%d mechanisms\n", count);
    }

    /* send capability list to client */
    send_string(out, data, len);
    dprintf(1, "waiting for client mechanism...\n");
    len = recv_string(in, chosenmech, sizeof chosenmech);
    if (len <= 0) {
        printf("client didn't choose mechanism\n");
        fputc('N', out); /* send NO to client */
        fflush(out);
        return -1;
    }

    if (mech && strcasecmp(mech, chosenmech)) {
        printf("client didn't choose mandatory mechanism\n");
        fputc('N', out); /* send NO to client */
        fflush(out);
        return -1;
    }

    len = recv_string(in, buf, sizeof(buf));
    if (len != 1) {
        saslerr(r, "didn't receive first-send parameter correctly");
        fputc('N', out);
        fflush(out);
        return -1;
    }

    /* additional code */
}

/* Additional code */

Appendix D • Source Code for SASL Example 257
if(buf[0] == 'Y') {
    /* receive initial response (if any) */
    len = recv_string(in, buf, sizeof(buf));

    /* start libasasl negotiation */
    r = sasl_server_start(conn, chosenmech, buf, len,
                          &data, &len);
} else {
    r = sasl_server_start(conn, chosenmech, NULL, 0,
                          &data, &len);
}

if (r != SASL_OK && r != SASL_CONTINUE) {
    saslerr(r, "starting SASL negotiation");
    fputc('N', out); /* send NO to client */
    fflush(out);
    return -1;
}

while (r == SASL_CONTINUE) {
    if (data) {
        dprintf(2, "sending response length %d...
", len);
        fputc('C', out); /* send CONTINUE to client */
        send_string(out, data, len);
    } else {
        dprintf(2, "sending null response...
");
        fputc('C', out); /* send CONTINUE to client */
        send_string(out, "", 0);
    }

dprintf(1, "waiting for client reply...
");
    len = recv_string(in, buf, sizeof buf);
    if (len < 0) {
        printf("client disconnected\n");
        return -1;
    }

    r = sasl_server_step(conn, buf, len, &data, &len);

    if (r != SASL_OK && r != SASL_CONTINUE) {
        saslerr(r, "performing SASL negotiation");
        fputc('N', out); /* send NO to client */
        fflush(out);
        return -1;
    }
}

if (r != SASL_OK) {
    saslerr(r, "incorrect authentication");
    fputc('N', out); /* send NO to client */
    fflush(out);
    return -1;
}

fputc('O', out); /* send OK to client */
    fflush(out);
    dprintf(1, "negotiation complete\n");

    r = sasl_getprop(conn, SASL_USERNAME, (const void **) &userid);
printf("successful authentication \"%s\n", userid);

    return 0;
}

int main(int argc, char *argv[]) {
    int c;
    char *port = "12345";
    char *service = "rcmd";
    int *l, maxfd=0;
    int r, i;
    sasl_conn_t *conn;

    while ((c = getopt(argc, argv, "p:s:m:")) != EOF) {
        switch(c) {
        case 'p':
            port = optarg;
            break;
        case 's':
            service = optarg;
            break;
        case 'm':
            mech = optarg;
            break;
        default:
            #ifdef _SUN_SDK_
            usage(argv[0]);
            #else
            usage();
            #endif /* _SUN_SDK_ */
            break;
        }
    }

    /* initialize the sasl library */
    r = sasl_server_init(callbacks, "sample");
    if (r != SASL_OK) saslfail(r, "initializing libsasl");

    /* get a listening socket */
    if ((l = listensock(port, PF_UNSPEC)) == NULL) {
        saslfail(SASL_FAIL, "allocating listensock");
    }

    for (i = 1; i <= l[0]; i++) {
        if (l[i] > maxfd)
            maxfd = l[i];
    }

    for (;;) {
        char localaddr[NI_MAXHOST | NI_MAXSERV],
            remoteaddr[NI_MAXHOST | NI_MAXSERV],
            hbuf[NI_MAXHOST], pbuf[NI_MAXSERV];
        struct sockaddr_storage local_ip, remote_ip;
        int salen;
    }
int nfds, fd = -1;
FILE *in, *out;
fd_set readfds;
FD_ZERO(&readfds);
for (i = 1; i <= l[0]; i++)
  FD_SET(l[i], &readfds);
nfds = select(maxfd + 1, &readfds, 0, 0, 0);
if (nfds <= 0) {
  if (nfds < 0 && errno != EINTR)
    perror("select");
  continue;
}
for (i = 1; i <= l[0]; i++)
  if (FD_ISSET(l[i], &readfds)) {
    fd = accept(l[i], NULL, NULL);
    break;
  }
if (fd < 0) {
  if (errno != EINTR)
    perror("accept");
  continue;
}
printf("accepted new connection\n");
/* set ip addresses */
salen = sizeof(local_ip);
if (getsockname(fd, (struct sockaddr *)&local_ip, &salen) < 0) {
  perror("getsockname");
}
getnameinfo((struct sockaddr *)&local_ip, salen,
hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),
#ifdef _SUN_SDK
  NI_NUMERICHOST | NI_NUMERICSERV);
#else
  NI_NUMERICHOST | NI_WITHSCOPEID | NI_NUMERICSERV);
#endif
snprintf(localaddr, sizeof(localaddr), "%s;%s", hbuf, pbuf);
salen = sizeof(remote_ip);
if (getpeername(fd, (struct sockaddr *)&remote_ip, &salen) < 0) {
  perror("getpeername");
}
getnameinfo((struct sockaddr *)&remote_ip, salen,
hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),
#ifdef _SUN_SDK
  NI_NUMERICHOST | NI_NUMERICSERV);
#else
  NI_NUMERICHOST | NI_WITHSCOPEID | NI_NUMERICSERV);
#endif
snprintf(remoteaddr, sizeof(remoteaddr), "%s;%s", hbuf, pbuf);
r = gethostname(myhostname, sizeof(myhostname)-1);
if(r == -1) saslfail(r, "getting hostname");
r = sasl_server_new(service, myhostname, NULL, localaddr, remoteaddr, NULL, 0, &conn);
if (r != SASL_OK) saslfail(r, "allocating connection state");
/* set external properties here
sasl_setprop(conn, SASL_SSF_EXTERNAL, &extprops); */
/* set required security properties here
sasl_setprop(conn, SASL_SEC_PROPS, &secprops); */
in = fdopen(fd, "r");
out = fdopen(fd, "w");
r = mysasl_negotiate(in, out, conn);
if (r == SASL_OK) {
    /* send/receive data */
}
printf("closing connection\n");
fclose(in);
fclose(out);
close(fd);
sasl_dispose(&conn);
}
sasl_done();

Common Code

The following code sample includes listings for miscellaneous SASL functions.

#pragma ident "@(#)common.c 1.1 03/03/28 SMI"
/* $Id: common.c,v 1.3 2002/09/03 15:11:59 rjs3 Exp $ */
/* Copyright (c) 2001 Carnegie Mellon University. All rights reserved. */
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
* 3. The name 'Carnegie Mellon University' must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For permission or any other legal
#include <config.h>
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#ifdef _SUN_SDK_
#include <sysexits.h>
#endif /* _SUN_SDK_ */
#include <sasl.h>

/* send/recv library for IMAP4 style literals.

really not important; just one way of doing length coded strings */

int send_string(FILE *f, const char *s, int l)
{
    int al;

    al = fprintf(f, "{%d}\r\n", l);
    fwrite(s, 1, l, f);
    fflush(f);

    printf("send: {%d}\n", l);
    while (l--)
    {
        if (isprint((unsigned char) *s)) {
            printf("%c", *s);
        } else {
            printf("%X", (unsigned char) *s);
        }
        s++;
    }
    printf("\n");
    return al;
}
int recv_string(FILE *f, char *buf, int buflen)
{
    int c;
    int len, l;
    char *s;

    c = fgetc(f);
    if (c != '{') return -1;

    /* read length */
    len = 0;
    c = fgetc(f);
    while (isdigit(c)) {
        len = len * 10 + (c - '0');
        c = fgetc(f);
    }
    if (c != '}') return -1;
    c = fgetc(f);
    if (c != '') return -1;
    c = fgetc(f);
    if (c != '
') return -1;

    /* read string */
    if (buflen <= len) {
        fread(buf, buflen - 1, 1, f);
        buf[buflen - 1] = '\0';
    } else {
        fread(buf, len, 1, f);
        buf[len] = '\0';
    }

    l = len;
    s = buf;
    printf("recv: {%d}\n", len);
    while (l--)
    {
        if (isprint((unsigned char) *s)) {
            printf("%c", *s);
        } else {
            printf("[%X]", (unsigned char) *s);
        }
        s++;
    }
    printf("\n");

    return len;
}

int debuglevel = 0;

int dprintf(int lvl, const char *fmt, ...)
{
    va_list ap;
    int ret = 0;

if (debuglevel >= lvl) {
    va_start(ap, fmt);
    ret = vfprintf(stdout, fmt, ap);
    va_end(ap);
}

    return ret;
}

void saslerr(int why, const char *what)
{
    fprintf(stderr, "%s: %s
", what, sasl_errstring(why, NULL, NULL));
}

void saslfail(int why, const char *what)
{
    saslerr(why, what);
    exit(EX_TEMPFAIL);
}
This appendix provides reference information for SASL, which is an acronym for simple authentication and security layer.

**SASL Interface Summaries**

The following tables provide brief descriptions of some SASL interfaces.

**TABLE E–1  SASL Functions Common to Clients and Servers**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_version</td>
<td>Get version information for the SASL library.</td>
</tr>
<tr>
<td>sasl_done</td>
<td>Release all SASL global state.</td>
</tr>
<tr>
<td>sasl_dispose</td>
<td>Dispose of sasl_conn_t when connection is done.</td>
</tr>
<tr>
<td>sasl_getprop</td>
<td>Get property, for example, user name, security layer info.</td>
</tr>
<tr>
<td>sasl_setprop</td>
<td>Set a SASL property.</td>
</tr>
<tr>
<td>sasl_errdetail</td>
<td>Generate string from last error on connection.</td>
</tr>
<tr>
<td>sasl_errstring</td>
<td>Translate SASL error code to a string.</td>
</tr>
<tr>
<td>sasl_encode</td>
<td>Encode data to send using security layer.</td>
</tr>
<tr>
<td>sasl_encodev</td>
<td>Encode a block of data for transmission through the security layer. Uses iovec * as the input parameter.</td>
</tr>
<tr>
<td>sasl_listmech</td>
<td>Create list of available mechanisms.</td>
</tr>
<tr>
<td>sasl_global_listmech</td>
<td>Return an array of all possible mechanisms. Note that this interface is obsolete.</td>
</tr>
<tr>
<td>sasl_seterror</td>
<td>Set the error string to be returned by sasl_errdetail().</td>
</tr>
</tbody>
</table>
### Table E-1  SASL Functions Common to Clients and Servers (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_idle</td>
<td>Configure <code>sasl</code> to perform calculations during an idle period or during a network round trip.</td>
</tr>
<tr>
<td>sasl_decode</td>
<td>Decode data received using security layer.</td>
</tr>
</tbody>
</table>

### Table E-2  Basic SASL Client-only Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_client_init</td>
<td>Called once initially to load and initialize client plug-ins.</td>
</tr>
<tr>
<td>sasl_client_new</td>
<td>Initialize client connection. Sets up the <code>sasl_conn_t</code> context.</td>
</tr>
<tr>
<td>sasl_client_start</td>
<td>Select mechanism for connection.</td>
</tr>
<tr>
<td>sasl_client_step</td>
<td>Perform one authentication step.</td>
</tr>
</tbody>
</table>

### Table E-3  Basic SASL Server Functions (Clients Optional)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_server_init</td>
<td>Called once initially to load and initialize server plug-ins.</td>
</tr>
<tr>
<td>sasl_server_new</td>
<td>Initialize server connection. Sets up the <code>sasl_conn_t</code> context.</td>
</tr>
<tr>
<td>sasl_server_start</td>
<td>Begin an authentication exchange.</td>
</tr>
<tr>
<td>sasl_server_step</td>
<td>Perform one authentication exchange step.</td>
</tr>
<tr>
<td>sasl_checkpass</td>
<td>Check a plain text passphrase.</td>
</tr>
<tr>
<td>sasl_checkapop</td>
<td>Check an APOP challenge/response. Uses a pseudo APOP mechanism, which is similar to a CRAM-MD5 mechanism. Optional. Note that this interface is obsolete.</td>
</tr>
<tr>
<td>sasl_user_exists</td>
<td>Check whether user exists.</td>
</tr>
<tr>
<td>sasl_setpass</td>
<td>Change a password. Optionally, add a user entry.</td>
</tr>
<tr>
<td>sasl_auxprop_request</td>
<td>Request auxiliary properties.</td>
</tr>
<tr>
<td>sasl_auxprop_getctx</td>
<td>Get auxiliary property context for connection.</td>
</tr>
</tbody>
</table>

### Table E-4  SASL Functions for Configuring Basic Services

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_set_alloc</td>
<td>Assign memory allocation functions. Note that this interface is obsolete.</td>
</tr>
<tr>
<td>sasl_set_mutex</td>
<td>Assign mutex functions. Note that this interface is obsolete.</td>
</tr>
</tbody>
</table>
### TABLE E-4  SASL Functions for Configuring Basic Services  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_client_add_plugin</td>
<td>Add a client plug-in.</td>
</tr>
<tr>
<td>sasl_server_add_plugin</td>
<td>Add a server plug-in.</td>
</tr>
<tr>
<td>sasl_canonuser_add_plugin</td>
<td>Add a user canonicalization plug-in.</td>
</tr>
<tr>
<td>sasl_auxprop_add_plugin</td>
<td>Add an auxiliary property plug-in.</td>
</tr>
</tbody>
</table>

### TABLE E-5  SASL Utility Functions  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_decode64</td>
<td>Use base64 to decode.</td>
</tr>
<tr>
<td>sasl_encode64</td>
<td>Use base64 to encode.</td>
</tr>
<tr>
<td>sasl_utf8verify</td>
<td>Verify that a string is valid UTF-8.</td>
</tr>
<tr>
<td>sasl_erasebuffer</td>
<td>Erase a security-sensitive buffer or password. Implementation might use</td>
</tr>
<tr>
<td></td>
<td>recovery-resistant erase logic.</td>
</tr>
</tbody>
</table>

### TABLE E-6  SASL Property Functions  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_clear()</td>
<td>Clear values and optionally requests from property context</td>
</tr>
<tr>
<td>prop_dispose()</td>
<td>Dispose of a property context</td>
</tr>
<tr>
<td>prop_dup()</td>
<td>Create new propctx which duplicates the contents of an existing propctx</td>
</tr>
<tr>
<td>prop_erase()</td>
<td>Erase the value of a property</td>
</tr>
<tr>
<td>prop_format()</td>
<td>Format the requested property names into a string</td>
</tr>
<tr>
<td>prop_get()</td>
<td>Return array of the propval structure from the context</td>
</tr>
<tr>
<td>prop_getnames()</td>
<td>Fill in an array of struct propval, given a list of property names</td>
</tr>
<tr>
<td>prop_new()</td>
<td>Create a property context</td>
</tr>
<tr>
<td>prop_request()</td>
<td>Add property names to a request</td>
</tr>
<tr>
<td>prop_set()</td>
<td>Add a property value to the context</td>
</tr>
<tr>
<td>prop_setvals()</td>
<td>Set the values for a property</td>
</tr>
<tr>
<td>sasl_auxprop_getctx()</td>
<td>Get auxiliary property context for connection</td>
</tr>
<tr>
<td>sasl_auxprop_request()</td>
<td>Request auxiliary properties</td>
</tr>
</tbody>
</table>
### TABLE E-7 Callback Data Types

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl_getopt_t</td>
<td>Get an option value. Used by both clients and servers.</td>
</tr>
<tr>
<td>sasl_log_t</td>
<td>Log message handler. Used by both clients and servers.</td>
</tr>
<tr>
<td>sasl_getpath_t</td>
<td>Get path to search for mechanisms. Used by both clients and servers.</td>
</tr>
<tr>
<td>sasl_verifyfile_t</td>
<td>Verify files for use by SASL. Used by both clients and servers.</td>
</tr>
<tr>
<td>sasl_canon_user_t</td>
<td>User name canonicalization function. Used by both clients and servers.</td>
</tr>
<tr>
<td>sasl_getsimple_t</td>
<td>Get user and language list. Used by clients only.</td>
</tr>
<tr>
<td>sasl_getsecret_t</td>
<td>Get authentication secret. Used by clients only.</td>
</tr>
<tr>
<td>sasl_chalprompt_t</td>
<td>Display challenge and prompt for response. Used by clients only.</td>
</tr>
<tr>
<td>sasl_getrealm_t</td>
<td>Get the authentication realm. Used by clients only.</td>
</tr>
<tr>
<td>sasl_authorize_t</td>
<td>Authorize policy callback. Used by servers only.</td>
</tr>
<tr>
<td>sasl_server_userdb_checkpass_t</td>
<td>Verify plain text password. Used by servers only.</td>
</tr>
<tr>
<td>sasl_server_userdb_setpass_t</td>
<td>Set plain text password. Used by servers only.</td>
</tr>
</tbody>
</table>

### TABLE E-8 SASL Include Files

<table>
<thead>
<tr>
<th>Include File</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl/saslplug.h</td>
<td></td>
</tr>
<tr>
<td>sasl/sasl.h</td>
<td>Needed for developing plug-ins</td>
</tr>
<tr>
<td>sasl/saslutil.h</td>
<td></td>
</tr>
<tr>
<td>sasl/prop.h</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE E-9 SASL Return Codes: General

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASL_BADMAC</td>
<td>Integrity check failed</td>
</tr>
<tr>
<td>SASL_BADVERS</td>
<td>Mismatch between versions of a mechanism</td>
</tr>
<tr>
<td>SASL_BADPARAM</td>
<td>Invalid parameter supplied</td>
</tr>
<tr>
<td>SASL_BADPROT</td>
<td>Bad protocol, cancel operation</td>
</tr>
<tr>
<td>SASL_BUFOVER</td>
<td>Overflowed buffer</td>
</tr>
<tr>
<td>SASL_CONTINUE</td>
<td>Another step is needed in authentication</td>
</tr>
</tbody>
</table>
### TABLE E–9 SASL Return Codes: General (Continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASL_FAIL</td>
<td>Generic failure</td>
</tr>
<tr>
<td>SASL_NOMECH</td>
<td>Mechanism not supported</td>
</tr>
<tr>
<td>SASL_NOMEM</td>
<td>Insufficient memory to complete operation</td>
</tr>
<tr>
<td>SASL_NOTDONE</td>
<td>Cannot request information until later in exchange</td>
</tr>
<tr>
<td>SASL_NOTINIT</td>
<td>SASL library not initialized</td>
</tr>
<tr>
<td>SASL_OK</td>
<td>Successful result</td>
</tr>
<tr>
<td>SASL_TRYAGAIN</td>
<td>Transient failure, for example, a weak key</td>
</tr>
</tbody>
</table>

### TABLE E–10 SASL Return Codes: Client-Only

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASL_BADSERV</td>
<td>Server failed mutual authentication step</td>
</tr>
<tr>
<td>SASL_INTERACT</td>
<td>Needs user interaction</td>
</tr>
<tr>
<td>SASL_WRONGMECH</td>
<td>Mechanism does not support requested feature</td>
</tr>
</tbody>
</table>

### TABLE E–11 SASL Return Codes: Server-Only

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASL_BADAUTH</td>
<td>Authentication failure</td>
</tr>
<tr>
<td>SASL_BADVERS</td>
<td>Version mismatch with plug-in</td>
</tr>
<tr>
<td>SASL_DISABLED</td>
<td>Account disabled</td>
</tr>
<tr>
<td>SASL_ENCRYPT</td>
<td>Encryption needed to use mechanism</td>
</tr>
<tr>
<td>SASL_EXPIRED</td>
<td>Passphrase expired and needs to be reset</td>
</tr>
<tr>
<td>SASL_NOAUTHZ</td>
<td>Authorization failure</td>
</tr>
<tr>
<td>SASL_NOUSER</td>
<td>User not found</td>
</tr>
<tr>
<td>SASL_NOVERIFY</td>
<td>User exists, but without verifier</td>
</tr>
<tr>
<td>SASL_TOOWEAK</td>
<td>Mechanism too weak for this user</td>
</tr>
<tr>
<td>SASL_TRANS</td>
<td>One-time use of a plain text password enables requested mechanism for user</td>
</tr>
<tr>
<td>SASL_UNAVAIL</td>
<td>Remote authentication server unavailable</td>
</tr>
</tbody>
</table>
### TABLE E-12  SASL Return Codes – Password Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASL_NOCHANGE</td>
<td>Requested change not needed</td>
</tr>
<tr>
<td>SASL_NOUSERPASS</td>
<td>User-supplied passwords not permitted</td>
</tr>
<tr>
<td>SASL_PWLOCK</td>
<td>Passphrase locked</td>
</tr>
<tr>
<td>SASL_WEAKPASS</td>
<td>Passphrase too weak for security policy</td>
</tr>
</tbody>
</table>
Packaging and Signing Cryptographic Providers

This appendix describes how to package Solaris cryptographic provider applications and modules. The following topics are covered:

- “Packaging Cryptographic Provider Applications and Modules” on page 271
- “Adding Signatures to Providers” on page 273

Packaging Cryptographic Provider Applications and Modules

In the Solaris operating system, application software is delivered in units that are called packages. A package is a collection of files that are required for the distribution and installation of a software product. Packages are usually designed and built by the application developer after the development of the application code is complete. For general information about packaging software applications, see Application Packaging Developer’s Guide.

Packaging a cryptographic provider has two additional requirements:

- The developer must supply input files that add the application to the configuration files that manage the cryptographic framework.
- The developer must supply an X.509 certificate to indicate compliance with the United States government’s export laws. For testing purposes, the certificate can be generated prior to obtaining U.S. government approval. A package must have approval and a signed provider to be shipped.

Complying with U.S. Government Export Laws

The United States government restricts the export of open cryptographic interfaces, which are also referred to as crypto-with-a-hole. Due to this restriction, all vendors of providers must obtain export approval from the U.S. government. The vendor needs to request a certificate from Oracle Corporation to indicate compliance with export laws. The vendor then signs the provider electronically and ships the software with the certificate.
In the export approval process, the strength of your encryption determines the countries in which the software can be used. The U.S. government defines two export categories for encryption products that are manufactured in the U.S.A.:

- **Retail encryption products** – Retail encryption products are permitted to be shipped to all countries except for designated nations that are considered to be security threats.

- **Non-retail encryption products** – Non-retail encryption products are permitted for domestic use only and to countries that have been exempted by the U.S. government.

If your provider has non-retail approval, you can make the provider eligible for retail approval. Retail approval can be obtained by disabling the use of your provider by certain callers such as IPsec. Oracle provides two different certificates in this case, for restricted and unrestricted use. You indicate this situation in the certificate request process, “To Request a Certificate for Signing a Provider” on page 274. In addition, a special activation file must be generated, signed, and shipped with the provider. See “To Generate an Activation File for Retail Export” on page 276.

### Packaging User-Level Provider Applications

A third-party developer of a user-level cryptographic provider application completes the following process:

1. Acquire a certificate from Oracle Corporation. Then, sign the library. See “Adding Signatures to Providers” on page 273.
2. Ship the certificate with the package. The certificate must be placed in the `/etc/crypto/certs` directory.
3. Add the `pkcs11conf` class into the CLASSES string of the `pkginfo` file. The following line should be added:
   ```
   CLASS=none pkcs11conf
   ```
4. Create an input file `pkcs11.conf` in the `etc/crypto` directory.
   The input file for user-level providers is named `pkcs11.conf`. This file specifies the path to the provider. The `pkcs11.conf` uses the following syntax for the entry:
   ```filename```
   The entry is an absolute path to a file such as `/opt/lib/$ISA/myProviderApp.so`. This file is added to the configuration file when `pkgadd` is run. Note the `$ISA` expression in the path name. `$ISA` points to either a 32-bit version or a 64-bit version of the application, as needed.
5. Add the following line to the package’s prototype file:
   ```
   e pkcs11conf etc/crypto/pkcs11conf 0644 root sys
   ```
Packaging Kernel-Level Provider Modules

A third-party developer of a kernel-level cryptographic provider module completes the following process:

1. Acquire a certificate from Oracle Corporation. Then, sign the kernel software module or device driver. See "Adding Signatures to Providers" on page 273.

2. Ship the certificate with the package. The certificate should be placed in the /etc/crypto/certs directory.

3. Add the kcfconf class into the CLASSES string of the pkginfo file. The following line should be added:
   
   \[
   \text{CLASS=} \text{none kcfconf}
   \]

4. Create an input file kcf.conf in the etc/crypto directory. This file is used to add software and hardware plug-ins to the kernel configuration file.
   
   - If the provider is a kernel software module with cryptographic mechanisms, use the following syntax for the entry:
     
     \[
     \text{provider-name: supported list=} \text{mech1, mech2, ...}
     \]
     
     \[
     \text{provider-name} \quad \text{Base name for the kernel software module}
     \]
     
     \[
     \text{mech*} \quad \text{Name of the cryptographic mechanism in the list}
     \]

   The following entry is an example of a kernel software module:

   \[
   \text{des: supported list=} \text{CKM DES_CBC, CKM DES ECB, CKM DES CFB}
   \]

   - If the provider is a device driver for cryptographic mechanisms, such as an accelerator card, then use the following syntax for the entry:

     \[
     \text{driver_names=} \text{devicedriver1, devicedriver2, ...}
     \]
     
     \[
     \text{devicedriver*} \quad \text{Name of a device driver for a cryptographic device}
     \]

   The following entry is an example of a device driver:

   \[
   \text{driver_names=} \text{dca}
   \]

Adding Signatures to Providers

This section describes how to add a digital signature to a provider so that the provider can work within the framework. The section also describes how to verify that a provider has been properly signed. Providers can be one of the following objects: a PKCS #11 library, a loadable kernel module that implements an algorithm, or a device driver for a hardware accelerator.
To Request a Certificate for Signing a Provider

Typically, the developer of a provider requests the certificate. However, the system administrator might be called on to handle the request as part of a site’s security policy.

1 Request a certificate from Sun by using the `elfsign request` command.

The command generates a private key along with the certificate request.

```
% elfsign request -k private-keyfile -r certificate-request
```

- `private-keyfile` Path to the location of the private key. This key is needed later when the system administrator signs providers for the Solaris cryptographic framework. The directory should be secure. Use a different directory from the directory that holds the Sun certificate.

- `certificate-request` Path to the certificate request.

The following example shows how a typical request is submitted to Sun:

```
% elfsign request \
-k /securecrypt/private/MyCompany.private.key \
-r /reqcrypt/MyCompany.certrequest
```

Enter Company Name / Stock Symbol or some other globally unique identifier. This will be the prefix of the Certificate DN: MYCORP

The government of the United States of America restricts the export of "open cryptographic interfaces", also known as "crypto-with-a-hole". Due to this restriction, all providers for the Solaris cryptographic framework must be signed, regardless of the country of origin.

The terms "retail" and "non-retail" refer to export classifications for products manufactured in the USA. These terms define the portion of the world where the product may be shipped. Roughly speaking, "retail" is worldwide (minus certain excluded nations) and "non-retail" is domestic only (plus some highly favored nations). If your provider is subject to USA export control, then you must obtain an export approval (classification) from the government of the USA before exporting your provider. It is critical that you specify the obtained (or expected, when used during development) classification to the following questions so that your provider will be appropriately signed.

Do you have retail export approval for use without restrictions based on the caller (for example, IPsec)? [Yes/No] N

If you have non-retail export approval for unrestricted use of your provider by callers, are you also planning to receive retail approval restricting which export sensitive callers (for example, IPsec) may use your provider? [Y/N] Y

The private key is placed in the file name that you specify, for example, `/etc/crypto/private/MyCompany.private.key` file. The certificate request is also placed in a file name that you specify, for example, `/reqcrypt/MyCompany.certrequest` file.
2 **Submit the certificate request to Sun.**

Send the certificate request to the following email address:
solaris-crypto-req_ww@oracle.com

Sun generates a certificate from your certificate request file. A copy of the certificate is sent back to you.

3 **Store the certificate that you receive from Sun in the `/etc/crypto/certs` directory.**

For security, the private key and the certificate request should be stored in other directories.

**To Sign a Provider**

Typically, the developer of the provider signs the provider. However, the system administrator might be called on to sign the developer’s binary as part of your site security policy.

- **Sign the provider. Use the `elfsign sign` command, the certificate from Sun, and the private key for requesting certificates from Sun.**

  ```
  % elfsign sign -k private-keyfile -c Sun-certificate -e provider-object
  -k     File that contains that private key that was used to generate the certificate request that was sent to Sun.
  -c     Path to the certificate from Sun that was issued from the certificate request.
  -e     Path to the provider, or binary, to be signed for use within the Solaris cryptographic framework.
  
  The following example shows how to sign a provider.
  
  % elfsign sign \
  -k /securecrypt/private/MyCompany.private.key \
  -c /etc/crypto/certs/MyCompany \
  -e /path/to/provider.object
  
  Note that using `elfsign sign` changes the object in the location that was specified. If an unsigned version of the object is needed, then the object should be copied to a different location before `elfsign sign` is applied.
  ```
To Verify That a Provider Is Signed

1. Collect the certificate that Sun issued and the path to the signed provider.

2. Verify that the provider is correctly signed by using the elfsign verify command.

   The following example demonstrates verification with the assumption that the certificate is in
   the default directory, /etc/crypto/certs/MyCompany.

   ```
   % elfsign verify \
   -e /path/to/MyProvider.so.1
   elfsign: verification of /path/to/MyProvider.so.1 passed
   ```

   The following example demonstrates storage of the certificate in a non-default directory.

   ```
   % elfsign verify \
   -c /path/to/MyCerts \
   -e /path/to/MyProvider.so.1
   elfsign: verification of /path/to/MyProvider.so.1 passed
   ```

   The following example demonstrates verification of a provider that has been signed with a
   restricted certificate.

   ```
   % elfsign verify \
   -e /path/to/MyRestrictedProvider.so.1
   elfsign: verification of /path/to/MyRestrictedProvider.so.1 passed, \n   but restricted.
   ```

To Generate an Activation File for Retail Export

This procedure is useful for when the same provider is to be shipped for both domestic use and
restricted international use. You sign the provider with a key for a usage-restricted certificate
for all customers. For those customers who use providers without caller-based restrictions, you
generate and include a special activation file that permits use with IPsec. The activation file
should reside in the same directory as the provider. The convention for naming the activation
file is to combine the name of the driver with the extension .esa, for example,
/kernel/drv/vca.esa.

- Sign the provider. Use the elfsign sign command, the certificate from Sun, and the private key
  for requesting certificates from Sun.

  ```
  % elfsign sign -a -k private-keyfile -c Sun-certificate -e provider-object
  ```

  Generate a signed ELF Sign Activation (.esa) file. This option is used when
  a cryptographic provider needs both non-retail export approval and retail
  approval. The retail approval is accomplished by restricting
  export-sensitive callers such as IPsec. This option assumes that the
  provider binary has previously been signed with a restricted certificate.
File that contains that private key that was used to generate the certificate request that was sent to Oracle Corporation.

Path to the certificate from Sun that was issued from the certificate request.

Path to the provider, or binary, to be signed for use within the Solaris cryptographic framework.

The following example shows how to sign a provider.

```
% elfsign sign \\
-a \\
-k /securecrypt/private/MyCompany.private.key \\
-c /etc/crypto/certs/MyCompany \\
-e /path/to/provider.object
```
<table>
<thead>
<tr>
<th><strong>Access Control List (ACL)</strong></th>
<th>A file containing a list of principals with certain access permissions. Typically, a server consults an access control list to verify that a client has permission to use its services. Note that a principal authenticated by GSS-API can still be denied services if an ACL does not permit them.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>authentication</strong></td>
<td>A security service that verifies the claimed identity of a principal.</td>
</tr>
<tr>
<td><strong>authorization</strong></td>
<td>The process of determining whether a principal can use a service, which objects the principal is allowed to access, and the type of access allowed for each.</td>
</tr>
<tr>
<td><strong>client</strong></td>
<td>Narrowly, a process that makes use of a network service on behalf of a user, for example, an application that uses rlogin. In some cases, a server can itself be a client of some other server or service. Informally, a principal that makes use of a service.</td>
</tr>
<tr>
<td><strong>confidentiality</strong></td>
<td>A security service that encrypts data. Confidentiality also includes integrity and authentication services. See also authentication, integrity, service.</td>
</tr>
<tr>
<td><strong>consumer</strong></td>
<td>An application, library, or kernel module that uses system services.</td>
</tr>
<tr>
<td><strong>context</strong></td>
<td>A state of trust between two applications. When a context has successfully been established between two peers, the context acceptor is aware that the context initiator is who it claims to be, and can verify and decrypt messages sent to it. If the context includes mutual authentication, then the initiator knows the acceptor's identity is valid and can also verify and decrypt messages from the acceptor.</td>
</tr>
<tr>
<td><strong>context-level token</strong></td>
<td>See token.</td>
</tr>
<tr>
<td><strong>credential</strong></td>
<td>An information package that identifies a principal and a principal's identification. A credential specifies who the principal is and, often, what privileges the principal has. Credentials are produced by security mechanisms.</td>
</tr>
<tr>
<td><strong>credential cache</strong></td>
<td>A storage space (usually a file) containing credentials stored by a given mechanism.</td>
</tr>
<tr>
<td><strong>data replay</strong></td>
<td>When a single message in a message stream is received more than once. Many security mechanisms support data replay detection. Replay detection, if available, must be requested at context-establishment time.</td>
</tr>
<tr>
<td><strong>data type</strong></td>
<td>The form that a given piece of data takes, for example, an int, a string, a gss_name_t structure, or a gss_OID_set structure.</td>
</tr>
</tbody>
</table>
If permitted by the underlying security mechanism, a principal (generally the context initiator) can designate a peer principal (usually the context acceptor) as a proxy by delegating its credentials to it. The delegated credentials can be used by the recipient to make requests on behalf of the original principal, as might be the case when a principal uses `rlogin` from machine to machine.

A mechanism name that has been converted from the GSS-API internal-name format to the GSS-API Exported Name format by `gss_export_name()`. An exported name can be compared with names that are in non-GSS-API string format with `memcmp()`. See also mechanism name (MN), name.

Historically, security flavor and authentication flavor were equivalent terms, as a flavor indicated a type of authentication, such as AUTH_UNIX, AUTH_DES, AUTH_KERB. RPCSEC_GSS is also a security flavor, even though it provides integrity and confidentiality services in addition to authentication.

The Generic Security Service Application Programming Interface. A network layer providing support for various modular security services. GSS-API provides for security authentication, integrity, and confidentiality services, and allows maximum portability of applications with regard to security. See also authentication, confidentiality, integrity.

A machine accessible over a network.

A security service that, in addition to user authentication, provides proof of the validity of transmitted data through cryptographic tagging. See also authentication, confidentiality, message integrity code (MIC).

A software package that specifies cryptographic techniques to achieve data authentication or confidentiality. Examples include Kerberos v5 and Diffie-Hellman public key.

A special instance of a GSS-API internal-format name. A normal internal-format GSS-API name can contain several instances of a name, each in the format of an underlying mechanism. A mechanism name, however, is unique to a particular mechanism. Mechanism names are generated by `gss_canonicalize_name()`.

Data in the form of a `gss_buffer_t` object that is sent from one GSS-API-based application to a peer. An example of a message is "ls" sent to a remote ftp server.

A message can contain more than just the user-provided data. For example, `gss_wrap()` takes an unwrapped message and produces a wrapped one to be sent. The wrapped message includes both the original message and an accompanying MIC. GSS-API-generated information that does not include a message is a token. See token.

A cryptographic tag that is attached to transmitted data to ensure the data's validity. The recipient of the data generates another MIC and compares this MIC to the one that was sent. If the MICs are equal, the message is valid. Some MICs, such as those generated by `gss_get_mic()`, are visible to the application, while others, such as those generated by `gss_wrap()` or `gss_init_sec_context()`, are not.

See token.

See message integrity code (MIC).

See mechanism name (MN).
When a context is established, a context initiator must authenticate itself to the context acceptor. In some cases the initiator might request that the acceptor authenticate itself back. If the acceptor does so, the two are said to be **mutually authenticated**.

**name**

The name of a principal, such as `user@machine`. Names in the GSS-API are handled through the `gss_name_t` structure, which is opaque to applications. See also exported name, mechanism name (MN), name type, principal.

**name type**

The particular form in which a name is given. Name types are stored as `gss_OID` types and are used to indicate the format used for a name. For example, the name `user@machine` would have a name type of `GSS_C_NT_HOSTBASED_SERVICE`. See also exported name, mechanism name (MN), name.

**opaque**

Applies to a piece of data whose value or format is not normally visible to functions that use it. For example, the `input_token` parameter to `gss_init_sec_context()` is opaque to the application, but significant to the GSS-API. Similarly, the `input_message` parameter to `gss_wrap()` is opaque to the GSS-API but important to the application doing the wrapping.

**out-of-sequence detection**

Many security mechanisms can detect whether messages in a message stream are received out of their proper order. Message detection, if available, must be requested at context-establishment time.

**per-message token**

See token.

**principal**

A uniquely named client/user or server/service instance that participates in a network communication; GSS-API-based transactions involve interactions between principals. Examples of principal names include:

- user
- user@machine
- nfs@machine
- 123.45.678.9

See also name, name type.

**privacy**

See confidentiality.

**provider**

An application, library, or kernel module that provides services to consumers.

**Quality of Protection (QOP)**

A parameter used to select the cryptographic algorithms to be used in conjunction with the integrity or confidentiality service. With integrity, the QOP specifies the algorithm for producing a message integrity code (MIC). With confidentiality, the QOP specifies the algorithm for both the MIC and message encryption.

**replay detection**

Many security mechanisms can detect whether a message in a message stream has been incorrectly repeated. Message replay detection, if available, must be requested at context-establishment time.

**security flavor**

See flavor.

**security mechanism**

See mechanism.

**security service**

See service.
server

A principal that provides a resource to network clients. For example, if you use `rlogin` to log in to the machine `boston.eng.acme.com`, then that machine is the server providing the `rlogin` service.

service

1. (Also, network service) A resource provided to network clients; often provided by more than one server. For example, if you use `rlogin` to log in to the machine `boston.eng.acme.com`, then that machine is the server providing the `rlogin` service.

2. A security service can be either integrity or confidentiality, providing a level of protection beyond authentication. See also authentication, integrity, and confidentiality.

token

A data packet in the form of a GSS-API `gss_buffer_t` structure. Tokens are produced by GSS-API functions for transfer to peer applications.

Tokens come in two types. Context-level tokens contain information used to establish or manage a security context. For example, `gss_init_sec_context()` bundles a context initiator’s credential handle, the target machine’s name, flags for various requested services, and possibly other items into a token to be sent to the context acceptor.

Message tokens (also known as per-message tokens or message-level tokens) contain information generated by a GSS-API function from messages to be sent to a peer application. For example, `gss_get_mic()` produces an identifying cryptographic tag for a given message and stores it in a token to be sent to a peer with the message. Technically, a token is considered to be separate from a message, which is why `gss_wrap()` is said to produce an `output_message` and not an `output_token`.

See also message.
Index

A
access control lists, use in GSS-API, 63
account management, PAM service module, 38
ACL, See access control list
acquiring context information, 83
anonymous authentication, 78–79
APDU, SCF, 182
authentication
flavor, 280
GSS-API, 57
anonymou, 78–79
mutual, 78
PAM process for, 39
PAM service module, 38
SASL, 131
authid
auxprop plug-ins, 144
SASL, 124
authorizations
code example, 34
defined, 23
use in application development, 34
authzid, auxprop plug-ins, 144
auxiliary properties, See auxprop plug-ins
auxprop plug-ins, 144

C
C_CloseSession() function (Continued)
random byte generation example, 177
C_Decrypt() function, 166
C_DecryptInit() function, 166
C_EncryptFinal() function, 166
C_EncryptInit() function, 166
C_EncryptUpdate() function, 166
C_Finalize() function
digest message example, 163
message signing example, 170
C_GenerateKeyPair() function, 170
C_GenerateRandom() function, 176
C_GetAttributeValue() function, 170
C_GetInfo() function, 157, 163
C_GetMechanismList() function, 160
C_GetSlotList() function, 158
message signing example, 170
random byte generation example, 176
C_Initialize() function, 157
C_OpenSession() function, 159
random byte generation example, 176
C_SignInit() function, 170
C_Verify() function, 170
C_VerifyInit() function, 170
callbacks
SASL, 125
SASL_CB_AUTHNAME, 126
SASL_CB_CANON_USER, 127
SASL_CB_ECHOPROMPT, 127
SASL_CB_GETCONF, 126
SASL_CB_GETOPT, 126
SASL_CB_GETPATH, 126
callbacks, SASL (Continued)
  SASL_CB_GETREALM, 127
  SASL_CB_LANGUAGE, 126
  SASL_CB_LOG, 126
  SASL_CB_NOECHOPROMPT, 127
  SASL_CB_PASS, 126
  SASL_CB_PROXY_POLICY, 127
  SASL_CB_SERVER_USERDB_CHECKPASS, 127
  SASL_CB_SERVER_USERDB SETPASS, 127
  SASL_CB_USER, 126
  SASL_CB VERIFYFILE, 126
canonicalization, SASL, 143
card objects, SCF, 182
certificates
  cryptographic applications, 271
  requesting from Sun, 274
channel bindings
  GSS-API, 79–80, 223
client establish context() function, GSS-API
  client example, 97
client plug-ins
  SASL, 141, 144
confidentiality
  GSS-API, 57, 83
connect_to_server() function
  GSS-API client example, 96, 100
connection contexts, SASL, 127
consumers
  defined, 21
  Solaris cryptographic framework, 147
context-level tokens, GSS-API, 69
contexts
  GSS-API
    acceptance, 75–77
    acceptance example, 115–119
    deletion, 91
    establishing, 73–83
    establishing example, 98
    exporting, 81
    getting acquisition information, 83
    gss-client example, 106
    import and export, 80–81, 120–121
    introduction, 56
    other context services, 77
credentials
  cache, 279
  delegation, 77–78
  GSS-API, 72–73, 226
  acquisition, 110
  GSS-API default, 72
crypto pseudo device driver, 150
cryptoadm pseudo device driver, 151
cryptoadm utility, 151
cryptographic checksum (MIC), 84–85
cryptographic framework, See Solaris cryptographic framework

cryptographic providers, Oracle Solaris cryptographic framework, 150
cryptoki library, overview, 155

D
data encryption, GSS-API, 85
data protection, GSS-API, 83
data replay, 279
data types
  GSS-API, 59–68, 221–224
    integers, 59
    names, 60–62
    strings, 59–60
    privileges, 27–28
default credentials, GSS-API, 72
delegation, credentials, 77–78
design requirements
  Oracle Solaris cryptographic framework
    kernel-level consumers, 152
    user-level providers, 151
  Solaris cryptographic framework
    user-level consumers, 151
digesting messages, Oracle Solaris cryptographic framework, 163
effective privilege set, defined, 25
elfsign command
  Oracle Solaris cryptographic framework, 151
  request subcommand, 274
  sign subcommand, 275, 276
  verify subcommand, 276
encryption
  GSS-API, 83
  wrapping messages with gss_wrap(), 85
encryption products, export laws, 271
error codes, GSS-API, 218
examples
  checking for authorizations, 34
  GSS-API client application
    description, 93
    source code, 187
  GSS-API miscellaneous functions
    source code, 207
  GSS-API server application
    description, 107
    source code, 197
miscellaneous SASL functions, 261
Oracle Solaris cryptographic framework
  message digest, 163
  random byte generation, 176
  signing and verifying messages, 170
  symmetric encryption, 165
PAM consumer application, 41
PAM conversation function, 45
PAM service provider, 50
privilege bracketing, 30
SASL client application, 245
SASL server application, 253
export laws, encryption products, 271
exporting GSS-API contexts, 80–81

functions (Continued)
  GSS-API, 215–217

G
General Security Standard Application Programming Interface, See GSS-API
GetMechanismInfo() function, 170
GetRandSlot() function, 176
GetTokenInfo() function, 176
gss_accept_sec_context() function, 75, 216
  GSS-API server example, 119
gss_acquire_cred() function, 72, 215
  GSS-API server example, 110
gss_add_cred() function, 73, 215
gss_add_oid_set_member() function, 216
GSS-API
  acquiring credentials, 110
  anonymous authentication, 78–79
  anonymous name format, 225
  channel bindings, 79–80, 223
  communication layers, 55
  comparing names in, 63–66
  confidentiality, 83
  constructing OIDs, 241–242
  context establishment example, 98
  contexts
    acceptance example, 115–119
    deallocation, 91
    expiration, 226
  createMechOid() function, 242
  credentials, 72–73
    expiration, 226
data types, 59–68, 221–224
detecting out-of-sequence problems, 87
developing applications, 71–91
displaying status codes, 220–221
displaying status codes, 220–221
encryption, 83, 85
exporting contexts, 81, 226
files containing OID values, 239–240
functions, 215–217
generalized steps, 71
gss_client example
  context deletion, 106
GSS-API, gss-client example (Continued)
   contexts, 102
   sending messages, 103
   signature blocks, 105

GSS-API (Continued)
   status codes, 68–69, 217–221
   supported credentials, 226
   tokens, 69–71
      context-level, 69
      interprocess, 70–71
      per-message, 69
   translation into GSS-API format, 98
   wrap-size limits, 226

   gss_buffer_desc structure, 59
   gss_buffer_desc structure, 222
   gss_buffer_t pointer, 60
   GSS_C_ACCEPT credential, 72
   GSS_C_BOTH credential, 72
   GSS_C_INITIATE credential, 72
   GSS_CALLING_ERROR macro, 68, 221
   gss_canonicalize_name() function, 61, 216
   gss_channel_bindings_structure structure, 222
   gss_channelBindings_t data type, 79

   gss-client example
      context deletion, 106
      obtaining context status, 102
      restoring contexts, 102
      saving contexts, 102
      sending messages, 103
      signature blocks, 105
   gss-client sample application, 93
   gss_compare_name() function, 63, 65, 216
   gss_context_time() function, 216
   gss_create_empty_oid_set() function, 217
   gss_delete_oid() function, 217
   gss_delete_sec_context() function, 91, 216
      releasing contexts, 225
   gss_display_name() function, 61, 216
   gss_display_status() function, 216, 220–221
   gss_duplicate_name() function, 216
   gss_export_context() function, 70
   gss_export_name() function, 216
   gss_export_sec_context() function, 80, 216
   gss_get_mic() function, 83, 84–85, 216
      comparison with gss_wrap() function, 83
   GSS-API server example, 120
   gss_import_name() function, 60, 216
   GSS-API client example, 98

   gss_server example
      signing messages, 120
      unwrapping messages, 119
   gss_str_to_oid() function, 240–241
   includefiles, 71
   integrity, 83
   interprocess tokens, 226
   introduction, 55–59
   Kerberos v5 status codes, 227
   language bindings, 59
   limitations, 58
   mech file, 239–240
   message transmission, 89
   MICs, 83
   minor-status codes, 226
   miscellaneous sample functions
      source code, 207
   mutual authentication, 78
   name types, 67–68, 222–223
   OIDs, 66–68
   other context services, 77
   outside references, 59
   portability, 56–57
   protecting channel-binding information, 225–226
   QOP, 57, 240
   readable name syntax, 225
   releasing contexts, 121
   releasing stored data, 225
   remote procedure calls, 57–58
   replaced functions, 217
   role in Solaris OS, 21
   sample client application
      description, 93
      source code, 187
   sample server application
      description, 107
      source code, 197
   specifying non-default mechanisms, 243
   specifying OIDs, 239
   status code macros, 221

   gss-server example
      signing messages, 120
      unwrapping messages, 119
   gss_str_to_oid() function, 240–241
   includefiles, 71
   integrity, 83
   interprocess tokens, 226
   introduction, 55–59
   Kerberos v5 status codes, 227
   language bindings, 59
   limitations, 58
   mech file, 239–240
   message transmission, 89
   MICs, 83
   minor-status codes, 226
   miscellaneous sample functions
      source code, 207
   mutual authentication, 78
   name types, 67–68, 222–223
   OIDs, 66–68
   other context services, 77
   outside references, 59
   portability, 56–57
   protecting channel-binding information, 225–226
   QOP, 57, 240
   readable name syntax, 225
   releasing contexts, 121
   releasing stored data, 225
   remote procedure calls, 57–58
   replaced functions, 217
   role in Solaris OS, 21
   sample client application
      description, 93
      source code, 187
   sample server application
      description, 107
      source code, 197
   specifying non-default mechanisms, 243
   specifying OIDs, 239
   status code macros, 221
gss_import_name() function (Continued)
    GSS-API server example, 111
    gss_import_sec_context() function, 80, 216
    gss_indicate_mechs() function, 216
    gss_init_sec_context() function, 73, 77, 215
    GSS-API client example, 98
    use in anonymous authentication, 78–79
    use in mutual authentication, 78
    gss_inquire_context() function, 216
    gss_inquire_context() function, 83
    gss_inquire_cred_by_mech() function, 215
    gss_inquire_cred() function, 215
    gss_inquire_mechs_for_name() function, 216
    gss_inquire_names_for_mech() function, 216
    gss_OID_desc structure, 222
    gss_OID pointer, 67
    gss_OID_set_desc structure, 67
    gss_OID_set_desc structure, 222
    gss_OID_set pointer, 67
    gss_to_str() function, 217
    gss_process_context_token() function, 216
    gss_release_buffer() function, 91, 217
    gss_release_cred() function, 91, 215
    GSS-API server example, 121
    gss_release_name() function, 91, 216
    releasing stored data, 225
    gss_release_oid() function
    GSS-API client example, 95
    GSS-API server example, 111
    gss_release_oid_set() function, 91, 217
    GSS_ROUTINE_ERROR macro, 68, 221
    gss_seal() function, 217
    gss_server example
    signing messages, 120
    unwrapping messages, 119
    gss_server sample application, 107
    gss_sign() function, 217
    gss_str_to_oid() function, 217, 240–241
    GSS_SUPPLEMENTARY_INFO macro, 68, 221
    gss_test_oid_set_member() function, 217
    gss_unseal() function, 217
    gss_unwrap() function, 216
    GSS-API server example, 119
    gss_verify() function, 217
    gss_verify_mic() function, 216
    gss_wrap() function, 83, 85, 216
    comparison with gss_get_mic() function, 83
    wrapping messages, 85
    gss_wrap_size_limit() function, 85, 216
    gssapi.h file, 71
    guidelines for privileged applications, 33
    header files, GSS-API, 71

<table>
<thead>
<tr>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFD handlers</td>
</tr>
<tr>
<td>developing for smart card terminals, 185</td>
</tr>
<tr>
<td>SCF architecture, 181</td>
</tr>
<tr>
<td>IFDHCloseChannel() function, 185</td>
</tr>
<tr>
<td>IFDHCreateChannelByName() function, 185</td>
</tr>
<tr>
<td>IFDHGetCapabilities() function, 185</td>
</tr>
<tr>
<td>IFDHICCPresence() function, 185</td>
</tr>
<tr>
<td>IFDHPowerICC() function, 185</td>
</tr>
<tr>
<td>IFDHSetProtocolParameters() function, 185</td>
</tr>
<tr>
<td>IFDHTransmitToICC() function, 185</td>
</tr>
<tr>
<td>importing GSS-API contexts, 80–81</td>
</tr>
<tr>
<td>inetd, checking for in gss-client() example, 113</td>
</tr>
<tr>
<td>inheritable privilege set, defined, 25</td>
</tr>
<tr>
<td>integers, GSS-API, 59</td>
</tr>
<tr>
<td>integrity</td>
</tr>
<tr>
<td>GSS-API, 57, 83</td>
</tr>
<tr>
<td>interprocess tokens, GSS-API, 70–71</td>
</tr>
<tr>
<td>IPC privileges, 26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java API, 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kerberos v5, GSS-API, 57</td>
</tr>
</tbody>
</table>
Index

L
language bindings, GSS-API, 59
libpam, 39
libpkcs11.so library, Solaris cryptographic framework, 150
libsasl
  initialization, 128
  use of API, 125
libsasl library, 123
limit privilege set, defined, 25
listener objects, SCF, 182

M
macros
  GSS-API
    GSS_CALLING_ERROR, 68
    GSS_ROUTINE_ERROR, 68
    GSS_SUPPLEMENTARY_INFO, 68
major status codes
  GSS-API, 68
  descriptions, 218
major-status codes
  GSS-API
    encoding, 217
mech file, 239–240
Mechanism Name (MN), 62
mechanisms
  defined, 21
  GSS-API, 57
  printable formats, 241
  SASL, 124
  Solaris cryptographic framework, 147
  specifying GSS-API, 67
memcmp function, 65
message digesting, Oracle Solaris cryptographic framework, 163
Message Integrity Code, See MICs
messages
  See also data
    encrypting with gss_wrap(), 85
    GSS-API, 69
    out-of-sequence problems, 87
    sending, 103
  signing, 120
  transmission confirmation, 89
  unwrapping, 119
  tagging with MICs, 84–85
  wrapping in GSS-API, 85
metaslot, Solaris cryptographic framework, 147
MICs
  defined, 83
  GSS-API
    tagging messages, 84–85
    message transmission confirmation, 89
minor status codes, GSS-API, 68
MN, See Mechanism Name
  mutual authentication, GSS-API, 78

N
name types, GSS-API, 222–223
names
  comparing in GSS-API, 63–66
  GSS-API, 60–62
  types in GSS-API, 67–68
network security, overview, 21
non-retail encryption products, export laws, 272

O
Object Identifiers, See OIDs
OIDs
  constructing, 241–242
  deallocation of, 67
  GSS-API, 66–68
  sets, 67
  specifying, 67, 239
  types of data stored as, 67
Oracle Solaris cryptographic framework
  crypto pseudo device driver, 150
cryptoadm pseudo device driver, 151
cryptoadm utility, 151
cryptographic providers, 150
cryptoki library, 155
Oracle Solaris cryptographic framework (Continued)

design requirements
  kernel-level consumers, 152
  special treatment of _fini() function, 152
  user-level providers, 151
elfsign utility, 151
examples
  message digest, 163
  random byte generation, 176
  signing and verifying messages, 170
  symmetric encryption, 165
kernel programmer interface, 150
libpkcs11.so, 150
modules verification library, 150
pkcs11_kernel.so, 150
pkcs11_softtoken.so, 150
plugable interface, 150
scheduler / load balancer, 150
out-of-sequence problems, GSS-API, 87

P

cryptographic applications, 271
crypto functions, 287
pluggable interface, 150
PKCS #11
  C_GetInfo() function, 157
  C_GetMechanismList() function, 160
  C_GetSlotList() function, 158
  C_GetTokenInfo() function, 158
  C_Initialize() function, 157
  C_OpenSession() function, 159
  function list, 156
  pkcs11_softtoken.so module, 155
  SUNW_C_GetMechSession() function, 162
pkcs11_kernel.so library, Oracle Solaris
  cryptographic framework, 150
pkcs11_softtoken.so library, Oracle Solaris
  cryptographic framework, 150
plug-ins
  SASL, 139
  Solaris cryptographic framework, 147
pluggable interface, Oracle Solaris cryptographic
  framework, 150
principals, GSS-API, 60
PRIV_FILE_LINK_ANY, 26
PRIV_OFF flag, 27
PRIV_ON flag, 27
PRIV_PROC EXEC, 26
PRIV_PROC FORK, 26
PRIV_PROC_INFO, 26
PRIV_PROC_SESSION, 26
PRIV_SET flag, 28
priv_set_t structure, 27
priv_str_to_set() function, synopsis, 29
priv_t_type, 27
privilege sets, defined, 24
privileged applications, defined, 23
privileges
  assignment, 24
  bracketing in the least privilege model, 30
  bracketing in the superuser model, 30
  categories, 26
  IPC, 26
privileges, categories (Continued)
   process, 26
   system, 27
   System V IPC, 26
code example, 30
compatibility with superuser, 26
data types, 27–28
defined, 23
interfaces, 28
introduction, 19
operation flags, 27
overview, 24
priv_str_to_set() function, 29
privilege ID data type, 27
required header file, 27
setppriv() function, 29
use in application development, 33
process privileges, 26
See privileges
protecting data, GSS-API, 83
providers
defined, 21
packaging kernel-level applications, 273
packaging user-level applications, 272
Solaris cryptographic framework, 147, 150

Q

qop file, 240
QOPs, 57
   role in wrap size, 85
   specifying, 67, 239–240
   storage in OIDs, 66
Quality of Protection, See QOP

R

random byte generation
   Oracle Solaris cryptographic framework example, 176
remote procedure calls, GSS-API, 57–58
retail encryption products, export laws, 272
return codes, GSS-API, 68–69

RPCSEC_GSS, 57–58

S

SASL
   architecture, 124
   authentication, 131
   authid, 124
   auxprop plug-ins, 144
callbacks
   SASL_CB_AUTHNAME, 126
   SASL_CB_CANON_USER, 127
   SASL_CB_ECHOPROMPT, 127
   SASL_CB_GETCONF, 126
   SASL_CB_GETOPT, 126
   SASL_CB_GETPATH, 126
   SASL_CB_GETREALM, 127
   SASL_CB_LANGUAGE, 126
   SASL_CB_LOG, 126
   SASL_CB_NOECHOPROMPT, 127
   SASL_CB_PASS, 126
   SASL_CB_PROXY_POLICY, 127
   SASL_CB_SERVER_USERDB_CHECKPASS, 127
   SASL_CB_SERVER_USERDB_SETPASS, 127
   SASL_CB_USER, 126
   SASL_CB_VERIFYFILE, 126
canonicalization, 143
client sample application, 245
confidentiality, 135
collection contexts, 127
dependencies, 265
integrity, 135
library, 123
libsasl API, 125
libsasl initialization, 128
life cycle, 128
mechanisms, 124
overview, 123
plug-in design, 144
   client plug-ins, 141
   overview, 139
   server plug-ins, 142
structures, 140
reference tables, 265
SASL (Continued)
releasing resources, 135
releasing sessions, 135
role in Solaris OS, 22
sample functions, 261
sample output, 135
server sample application, 253
session initialization, 129
setting SSF, 129
SPI, 139
SSF, 125
userid, 124
sasl_canonuser_plug_nit() function, 143
SASL_CB_AUTHNAME callback, 126
SASL_CB_CANON_USER callback, 127
SASL_CB_ECHOPROMPT callback, 127
SASL_CB_GETCONF callback, 126
SASL_CB_GETOPT callback, 126
SASL_CB_GETREALM callback, 127
SASL_CB_LANGUAGE callback, 126
SASL_CB_LOG callback, 126
SASL_CB_NOECHOPROMPT callback, 127
SASL_CB_PASS callback, 126
SASL_CB_PROXY_POLICY callback, 127
SASL_CB_SERVER_USERDB_CHECKPASS callback, 127
SASL_CB_SERVER_USERDB_SETPASS callback, 127
SASL_CB_USER callback, 126
SASL_CB_VERIFYFILE callback, 126
sasl_client_add_plugin() function, 139
sasl_client_init() function, 128, 140
sasl_client_new() function, SASL life cycle, 129
sasl_client_start() function, SASL life cycle, 131
SASL_CONTINUE flag, 131
sasl_decode() function, 135
sasl_dispose() function, 135
sasl_done() function, 135
sasl_encode() function, 135
sasl_getprop() function, checking SSF, 135
SASL_INTERACT flag, 131
SASL_OK flag, 131
sasl_server_add_plugin() function, 139
sasl_server_init() function, 128, 140
sasl_server_new() function, SASL life cycle, 129
sasl_server_start() function, SASL life cycle, 131
SCF
card objects, 182
interfaces, 182
listener objects, 182
miscellaneous functions, 184
overview, 181
session functions, 183
session objects, 182
terminal functions, 183
terminal objects, 182
SCF_Card_close() function, 184
SCF_Card_exchangeAPDU() function, 184
SCF_Card_getInfo() function, 184
SCF_Card_lock() function, 184
SCF_Card_reset() function, 184
SCF_Card_unlock() function, 184
SCF_Card_waitForCardRemoved() function, 184
SCF_Session_close() function, 183
SCF_Session_freeInfo() function, 183
SCF_Session_getInfo() function, 183
SCF_Session_getSession() function, 183
SCF_Session_getTerminal() function, 183
SCF_strerror() function, 185
SCF_Terminal_addEventListener() function, 184
SCF_Terminal_close() function, 183
SCF_Terminal_freeInfo() function, 183
SCF_Terminal_getCard() function, 184
SCF_Terminal_getInfo() function, 183
SCF_Terminal_removeEventListener() function, 184
SCF_Terminal_updateEventListener() function, 184
SCF_Terminal_waitForCardAbsent() function, 184
SCF_Terminal_waitForCardPresent() function, 183
SEAM, GSS-API, 57
security context, See contexts
security flavor, 280
security mechanisms, See GSS-API
security policy, privileged application guidelines, 33
security strength factor, See SSF
send_token() function, GSS-API client example, 100
sequence problems, GSS-API, 87
291
server_acquire_creds() function, GSS-API server example, 110
server_establish_context() function, GSS-API server example, 115
server plug-ins, SASL, 142
service provider interface, See SPI
session management, PAM service module, 38
session objects
   SCF, 182
   Solaris cryptographic framework, 148
setppriv() function, synopsis, 29
shell escapes, and privileges, 34
sign_server() function
   GSS-API client example, 108
   GSS-API server example, 113
signature blocks
   GSS-API
      gss-client example, 105
signing messages, GSS-API, 120
signing messages example, Oracle Solaris cryptographic framework, 170
signing packages, 273
Simple Authentication and Security Layer, See SASL
slots, Solaris cryptographic framework, 147
Smart Card, role in Solaris OS, 22
smart card framework, See SCF
smart card terminals, installation guidelines, 186
soft tokens, Solaris cryptographic framework, 147
Solaris cryptographic framework
   architecture, 148
   design requirements
      user-level consumers, 151
      introduction, 147
      packaging applications, 271
      role in Solaris OS, 20
Solaris Enterprise Authentication Mechanism, See SEAM
Solaris smart card framework, See SCF
specifying a QOP, 239–240
specifying mechanisms in GSS-API, 239–240
specifying OIDs, 239
SPI
   Solaris cryptographic framework
      user level, 150
SSF
   defined, 125
   setting, 129, 131
status codes
   GSS-API, 68–69, 217–221
   major, 68
   minor, 68
strings, GSS-API, 59–60
SUNW_C_GetMechSession() function, 162
digest message example, 163
symmetric encryption example, 166
symmetric encryption
   Oracle Solaris cryptographic framework example, 165
system privileges, 27
System V IPC privileges, 26

T
terminal objects, SCF, 182
test_import_export_context() function, GSS-API server example, 120–121
token objects, Solaris cryptographic framework, 148
tokens
   distinguishing GSS-API types, 70
   GSS-API, 69–71
      context-level, 69
      interprocess, 70–71
      per-message, 69
   Solaris cryptographic framework, 147

U
userid, SASL, 124

V
verifying messages example
   Oracle Solaris cryptographic framework example, 170
Index

**W**
wrapping messages, GSS-API, 85

**X**
X.509 certificates, 271