The Solaris™ Cryptographic Framework

Paul Sangster, Valerie Bubb, Kais Belgaied

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.
March 2005

©2005 Sun Microsystems, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California.

Sun, Sun Microsystems, the Sun logo, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence,
s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie.

Sun, Sun Microsystems, le logo Sun, et Solaris sont des marques de fabrique ou des marques déposées, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOITPAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Table of Contents

1.0 Overview....4
2.0 Framework Architecture....4
2.1 Cryptographic Applications....6
2.2 Cryptographic Security Services....6
2.3 Framework....6
2.4 Cryptography Providers....7
3.0 Component Details....7
3.1 Applications....7
  3.1.2 Included End-User Commands....8
3.2 Framework....8
  3.2.1 User-Level Cryptographic Framework (uCF)....9
    3.2.1.1 Convenience Functions....9
  3.2.2 Kernel-Level Cryptographic Framework (kCF)....10
3.3 Module Verification Daemon....11
  3.3.1 Signing Providers....11
  3.4.1 Sun Software Crypto....11
  3.4.2 Sun Kernel Crypto Plug-In....12
3.5 Administration....12
3.6 Consumers....13
4.0 Distribution....13
5.0 References....13
Appendix A: User-Level Programming Example....14

1.0 Overview

The Solaris Cryptographic Framework provides cryptographic services to users and applications through commands, a user-level programming interface, a kernel programming interface, and user-level and kernel-level frameworks. The Solaris Cryptographic Framework provides these cryptographic services to applications and kernel modules in a manner seamless to the end user, and brings direct cryptographic services, like encryption and decryption for files, to the end user.

The user-level framework is responsible for providing cryptographic services to consumer applications and the end-user commands. The kernel-level framework provides cryptographic services to kernel modules and device drivers. Both frameworks give developers and users access to software-optimized cryptographic algorithms.

The programming interfaces are front-ends to each of the frameworks. A library or a kernel module that provides cryptographic services can be plugged into one of the frameworks by the system administrator, making the plug-in's cryptographic services available to applications or kernel modules. This flexibility allows the system administrator to plug in different cryptographic algorithm implementations or hardware-accelerated cryptographic providers.

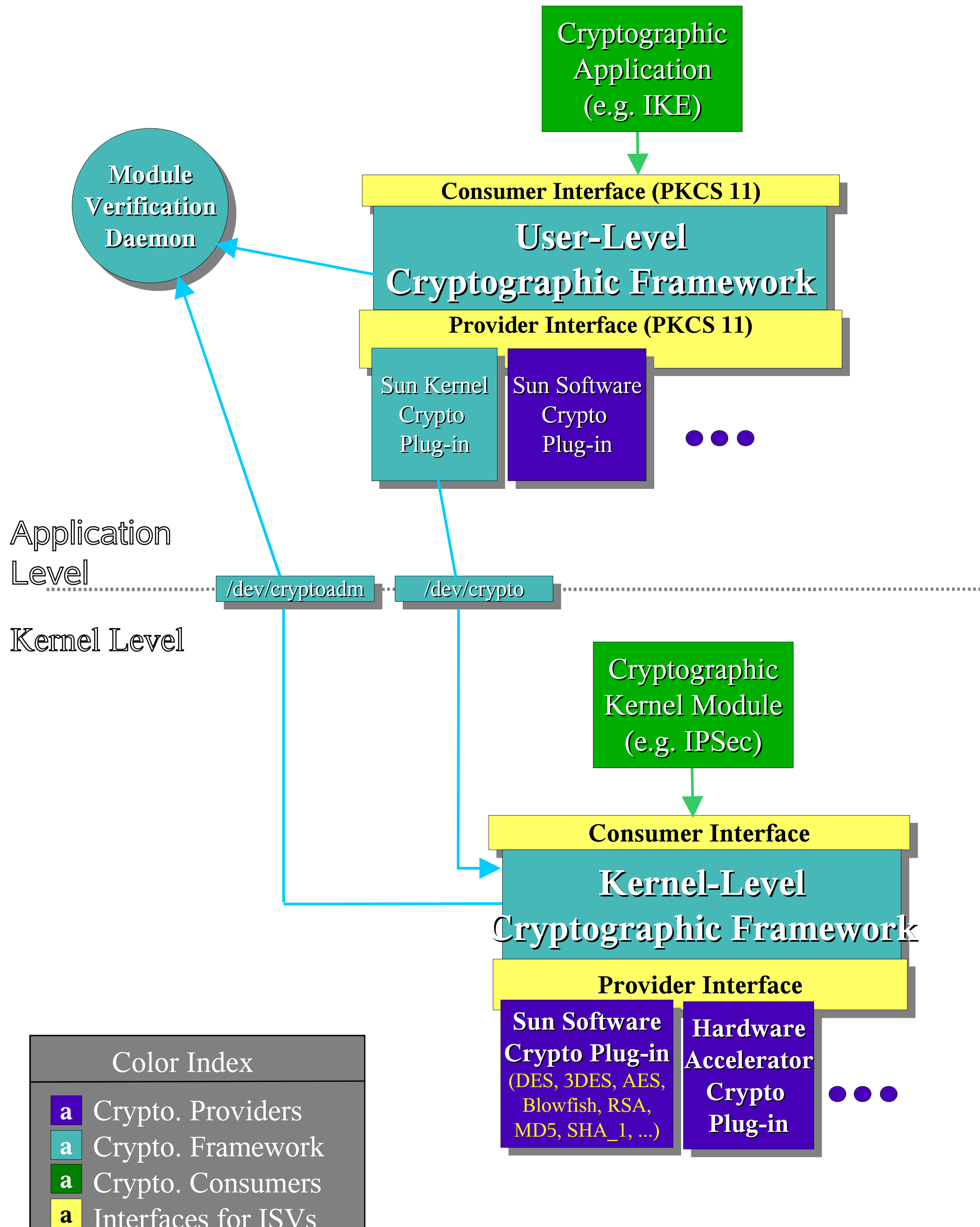
2.0 Framework Architecture

This section provides an overview of the Solaris Cryptographic Framework's architecture. The following diagram illustrates this architecture and the interrelationships between its many components and the Solaris Operating System (OS).

This architectural illustration makes use of the following conventions:

- Each software component is shown as a rectangle.
- Rectangle size does not indicate the size or complexity of the feature.
- Public (documented) programmatic interfaces appear as labeled bars on the top or bottom of the component. Bars at the top of a software module represent the APIs. Bars at the bottom represent the service provider interface (SPI).
- Plug-ins are represented as rectangles plugged into SPIs. These can be replaced or added to a framework. Third parties can only write such components if a public framework provider interface exists.
- Arrows indicate general (not literal) flow of control.
- Circles represent a Solaris process sometimes called a *daemon*.

Solaris Cryptographic Architecture



2.1 Cryptographic Applications

Components found in the application layer generally are not intended to provide generalized features to other applications, but rather perform a single class of function. An example application would be a Sendmail mail server whose role is to route the transfer of an email message towards its recipient. The mail server may make use of cryptography to protect the message as it traverses a hop to the next mail server. However the mail server does not provide protection for any other type of communication between the nodes.

In the cryptographic architecture of the Solaris OS, applications do not directly contain cryptographic algorithms (for example, DES). Instead they just make use of features of the *Cryptographic Security Services* available from the Solaris OS. This reduces code redundancy and provides access to optimized algorithms for all applications.

2.2 Cryptographic Security Services

The Solaris OS contains a variety of security services that make indirect use of cryptography. A high-level library offers a generalized interface for authenticating entities communicating across the network and optionally message integrity and privacy. Similar to the application components, these security services do not contain cryptographic algorithms, but rather leverage a common set of cryptography frameworks described below.

2.3 Framework

The framework components provide an abstracted, consistent interface to the *Cryptographic Security Services* for use of cryptography. These interfaces intend to make it easier for operating system components to integrate strong security into their offerings. An example framework component is the *User-Level Cryptographic Framework*. This framework offers an RSA Security Inc. PKCS #11 Cryptographic Token Interface (Cryptoki) standards-based [1] programmatic interface to security services, and a pluggable interface where appropriately signed cryptographic providers can register and provide services via the framework.

Similar to the *Cryptographic Security Service* components, the framework components do not directly include cryptography, but do offer a generalized programmatic interface to other parts of the operating system.

2.4 Cryptography Providers

The components at this bottom layer offer the actual cryptographic algorithms. For example, this layer includes software implementations of 3DES, AES, RSA, and SHA-1 just to name a few in the *Sun Software Crypto Plug-in*. These providers offer a PKCS #11-compliant user-level interface and PKCS #11-like kernel-level interface that is designed to plug into the cryptographic frameworks.

Since it is possible for third parties to write new or replace existing algorithms used by the Solaris Cryptographic Framework, the framework validates that each provider has an appropriate, embedded cryptographic signature. This signature limits who can write a provider that the framework recognizes and allows to operate, and the signature potentially limits the consumers who can make use of it.

3.0 Component Details

This section details the major components of the Solaris Cryptographic Framework.

3.1 Applications

Applications that currently employ PKCS#11 to access cryptography can modify their configuration to use the *User-Level Cryptographic Framework* out of the box. New applications can be linked to the library, `/usr/lib/libpkcs11.so`, to get direct access to the PKCS#11 functionality.

See section 3.2.1 for more details on the API for the *User-Level Cryptographic Framework.*

3.1.2 Included End-User Commands

The Solaris Cryptographic Framework provides a set of end-user commands that use the *User-Level Cryptographic Framework* and give access to cryptographic services to general users:

- `digest`(1) and `mac`(1), for calculating digest and MAC of files
- `encrypt`(1) and `decrypt`(1), for encrypting and decrypting files

3.2 Framework

This section focuses on the core Solaris Cryptographic Framework and its associated components. The cryptographic framework is responsible for managing a set of cryptographic providers that are plugged into its provider interface. Many of these providers will come from Sun (see section 3.4), but third parties are able to write plug-ins for the framework. In order to address crypto-with-a-hole exportability concerns, the framework requires cryptographic providers to be signed using an RSA private key associated with a certificate (public key) issued by Sun's Cryptographic Framework CA. This signature is validated before any use of the provider is allowed.

Except for signature validation of configured providers performed by the module verification daemon, the cryptographic framework does not include any cryptography. The signature validation process involves:

- Extracting the provider's signature for the library.
- Using RSA (public key) to decrypt the signature.
- Verifying that the result matches the MD-5 digest of the provider itself.

This operation prevents parties without a valid key from developing and signing a provider that is usable with the cryptographic framework.

The cryptographic framework comprises two frameworks with their respective providers and a user-level daemon that performs the verification of the providers registering for use with the frameworks.

3.2.1 User-Level Cryptographic Framework (uCF)

The first framework exists outside the kernel (at the user process level) and is intended for use by most consumers. This framework is referred to as the User-Level Cryptographic Framework, or uCF for short. The uCF offers a PKCS #11 v2.11-based API to callers wishing to have cryptographic operations performed. PKCS #11 was chosen for the User-Level Cryptographic Framework API because of the large volume of existing software that already supports this interface and the large number of developers that already understand this interface. See Appendix A for a uCF programming sample.

Below the API is mostly logic intended to glue the many cryptographic providers together and make them appear as a single aggregate provider. The uCF exposes a list of cryptographic "slots" (as they are known in PKCS #11) that enable the application to select the provider that best suits the application's particular needs.

At the bottom of the uCF is the pluggable cryptography provider interface. This interface is what cryptographic providers must implement in order to be usable with uCF. This provider interface also follows the PKCS #11 v2.11 interface, with the additional requirement that any provider that is configured to plug in must include a cryptographic signature. The signature is associated with an RSA key included in a certificate issued by Sun. This framework uses the *Module Verification Daemon* to verify that the provider is authorized for use, and to determine if there are usage restrictions on the provider. See section 3.3 for details on this daemon.

3.2.1.1 Convenience Functions

In addition to the standard PKCS #11 interfaces, Sun has provided two convenience functions that assist in the setup of PKCS #11 and with object creation. These functions do not exist in other implementations of PKCS #11, but rather are part of uCF.

```
CK_RV SUNW_C_GetMechSession(CK_MECHANISM_TYPE mech,
        CK_SESSION_HANDLE_PTR hSession);
```

Calling `SUNW_C_GetMechSession()` initializes the framework and does all of the necessary PKCS #11 calls to create a session that is capable of providing operations on the requested mechanism. It is not necessary to call `C_Initialize()` or `C_GetSlotList()` before the first call to `SUNW_C_GetMechSession()`.

If the function is called multiple times, it returns a new session without reinitializing the framework. If it is unable to return a new session, then CKR_SESSION_COUNT is returned.

C_CloseSession() should be used to release the session when it is no longer required.

```
CK_RV SUNW_C_KeyToObject(CK_SESSION_HANDLE hSession,
        CK_MECHANISM_TYPE mech, const void *rawkey,
        size_t rawkey_len, CK_OBJECT_PTR obj)
```

Calling SUNW_C_KeyToObject() creates a symmetric key object for the given mechanism from the rawkey data. rawkey is a pointer to any buffer. The value of rawkey is used to set the CKA_VALUE attribute in the resulting object. This function is useful if the programmer does not want to take all the steps necessary to set up a key template. The object should be destroyed with C_DestroyObject() when it is no longer required.

For more detailed information on the uCF API, please see the RSA PKCS #11 web site [1].

3.2.2 Kernel-Level Cryptographic Framework (kCF)

The other framework associated with the Solaris Cryptographic Framework exists within the kernel boundary and is known as the Kernel-Level Cryptographic Framework, or kCF. The kCF contains similar functionality to the uCF. The kCF offers a consumer and provider interface that is more applicable for kernel modules, such as cryptographic modules and device drivers from Sun and from third parties. The kCF can be used for Sun and third-party cryptographic modules and device drivers. The kCF contains no cryptography itself, not even for performing provider verification. This is because verification of each kernel-level provider is delegated to the *Module Verification Daemon* that executes at the user level and uses the same verification code as is used by uCF.

Some differences exist between the frameworks. The kCF not only offers a consumer and provider interface, but it also has two private device drivers, /dev/crypto and /dev/cryptoadm. uCF uses /dev/crypto (via a provider) to communicate with kCF. The module verification daemon and the cryptoadm(1M) command use /dev/cryptoadm to communicate with kCF. Additionally, kCF provides scheduling and load balancing for cryptographic operations for the kernel consumer, as well as offers an asynchronous mode for the consumer interface routines.

If you would like to take advantage of the kCF API for accessing optimized cryptographic algorithms and optional hardware, or if you would like to provide your own cryptographic services via the kCF SPI, please contact solaris-crypto-api@sun.com.

3.3 Module Verification Daemon

The module verification daemon has two main responsibilities. The first is to help set up the thread pool that lives in the kCF and is used to service requests. The other job is to answer requests for verification of user and kernel-level provider signatures. Verification assures that all providers are correctly signed by an authorized provider-signing key and to determine the provider's usage restrictions.

This process prevents unauthorized parties from writing providers usable with the cryptographic framework. This process is required to meet U.S. export regulations.

3.3.1 Signing Providers

A command, `elfsign`(1), is provided for generating a request form for certificates from Sun's Cryptographic Framework CA. Once the certificate is received, the same command can be used to sign and verify the binaries.

3.4 Cryptography Providers

This section discusses the cryptography that is present in the Sun-developed providers for the Solaris Cryptographic Framework (both at the user level and at the kernel level).

3.4.1 Sun Software Crypto

The Solaris OS includes two software crypto providers: one for uCF and the other for kCF. These providers bundle together a number of cryptographic algorithms and digests under a single PKCS #11 interface.

The following algorithms are supported by the *User-Level Cryptographic Framework* provider, `pkcs11_softtoken.so`.

Typical Cryptographic Use	Algorithm Names
Authentication	RSA, DSA, Diffie-Hellman, HMAC-MD5, HMAC SHA-1
Digesting	MD-5, SHA-1
Data Privacy	AES, DES, 3DES, RC4

The following algorithms are provided by kCF to consumers.

Typical Cryptographic Use	Algorithm Names
Authentication	RSA, HMAC-MD5, HMAC SHA-1
Digesting	MD-5, SHA-1
Data Privacy	AES, DES, 3DES, Blowfish, RC4

The goal of the software providers is to offer a base set of optimized software cryptographic algorithms so that Solaris components can rely on these being available, and their re-implementation can be avoided. Customers are given the ability to define policy, and this limits which of these algorithms can be used or allows them to remove a provider entirely.

3.4.2 Sun Kernel Crypto Plug-In

The uCF contains a provider that is included purely to facilitate communication with the kCF, `pkcs11_kernel.so`. This provider does not contain any cryptography itself, but appears to the uCF as offering all the cryptographic algorithms offered by the kCF. It is expected that this provider will primarily be used when a customer has installed a hardware cryptographic device that includes a provider for kCF. This provider will make the device's cryptographic algorithms available to uCF, which in turn can make them available to user-level applications.

3.5 Administration

One administration tool, `cryptoadm`(1M), is provided for administration of both uCF and kCF. The tool can be used for installing new hardware or software providers, defining policy for providers, and displaying cryptographic provider information.

3.6 Consumers

Solaris IPsec/IKE and Kerberos, user-level and kernel-level, have been ported to use the Solaris Cryptographic Framework in the Solaris 10 OS, gaining access to optimized cryptographic algorithms and hardware acceleration when available.

Solaris IPsec already allows for third-party key management daemons to be added, replacing the Solaris IKE implementations. This is useful in cases where Solaris IKE is not desired, which is possible due to legacy key management requirements. Solaris IPsec offers the PF_KEY (RFC 2367 [2]) API to the key management software for installation for keys and/or security associations.

4.0 Distribution

The Solaris Cryptographic Framework is available in the Solaris 10 Operating System (http://www.sun.com/software/solaris/index.jsp) and Solaris Express releases (http://www.sun.com/software/solaris/solaris-express/).

Information regarding the API or SPI for the Kernel Cryptographic Framework can be obtained by sending email to solaris-crypto-api@sun.com.

5.0 References

Note: Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

[1] RSA Laboratories, "PKCS #11 - Cryptographic Token Interface Standard," March 2001 (presently version 2.11)

http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html

[2] McDonald, D. L., Metz, C., Phan, B. G., "PF_KEY Key Management API, Version 2", July 1998
http://www.ietf.org/rfc/rfc2367.txt

Appendix A: User-Level Programming Example

This simple program reads in a file and computes the digest of it. This program should be linked with `libpkcs11.so` on a system that supports the Solaris Cryptographic Framework. This program uses one of the convenience functions detailed in section 3.2.1, so it is nonportable. It is possible to replace the convenience function with a series of PKCS #11 calls to make this portable.

Copyright 2005 Sun Microsystems, Inc. ALL RIGHTS RESERVED
Use of this software is authorized pursuant to the
terms of the license found at
http://developers.sun.com/berkeley_license.html

```
#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;

      /* Set the digest mechanism to target mechanism */
      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.  This same task could be
 * be performed with a series of PKCS #11 calls.
 */
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 digest file */
if ((fs = fopen(argv[1], "r")) == NULL) {
      perror("fopen");
      fprintf(stderr, "\n\tusage: %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,
                        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);

}
```