Managing Encryption and Certificates in Oracle® Solaris 11.3

Part No: E54783
October 2017

Managing Encryption and Certificates in Oracle Solaris 11.3

Part No: E54783

Contents

Using This Documentation

- **Overview** – Describes how to administer encryption, keys, and public/private certificates on one or more Oracle Solaris systems.
- **Audience** – System administrators who must implement security on the enterprise.
- **Required knowledge** – Familiarity with security concepts and terminology.

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394-01`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

Cryptography in Oracle Solaris

This chapter describes the central sources of cryptography in Oracle Solaris, and covers the following topics:

- "What's New in Cryptography for Oracle Solaris 11.3" on page 9
- "About Cryptography in Oracle Solaris" on page 10
- "OpenSSL and Oracle Solaris" on page 17

To administer and use cryptography, see Chapter 3, "Using the Cryptographic Framework" and "OpenSSL and Oracle Solaris" on page 17.

What's New in Cryptography for Oracle Solaris 11.3

This section highlights information for existing customers about new features in encryption support in this release.

- You can manually administer the Oracle Solaris keystore for Certificate Authority (CA) certificate files. For more information, see "Adding CA Certificates to the Oracle Solaris CA Keystore" on page 74.
- Your PKCS #11 applications can now function as clients that use the Key Management Interoperability Protocol (KMIP). These client applications can communicate with KMIP-compliant servers to create and use symmetric keys. Oracle Solaris provides client support for KMIP version 1.1, enabling clients to communicate with KMIP-compliant servers such as the Oracle Key Vault (OKV).

  For information, see Chapter 5, "KMIP and PKCS #11 Client Applications".

About Cryptography in Oracle Solaris

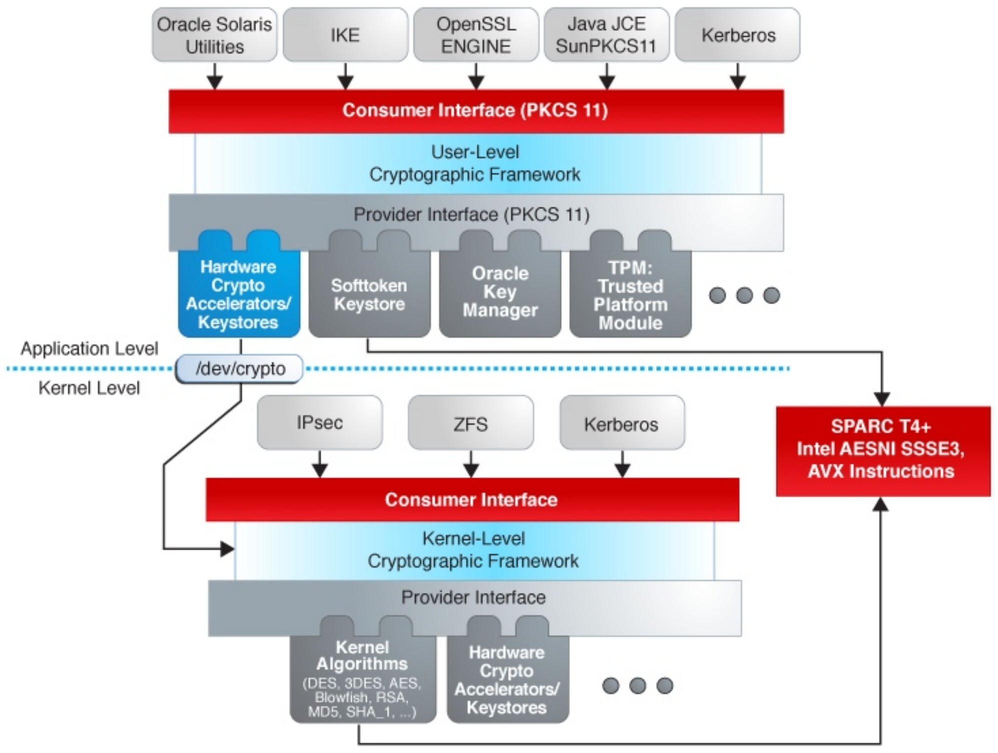
Oracle Solaris provides the Cryptographic Framework to handle cryptographic requirements. Third parties can add their cryptographic services as plugins to the Cryptographic Framework. See “Introduction to the Cryptographic Framework” on page 10.

OpenSSL, an open source project, is a source of cryptographic services for Transport Layer Security (TLS) and Secure Sockets in Oracle Solaris. Oracle Solaris supports both the non-FIPS 140-2 and the FIPS 140-2 versions of OpenSSL. See “OpenSSL and Oracle Solaris” on page 17.

Introduction to the Cryptographic Framework

The Cryptographic Framework provides a common store of algorithms and PKCS #11 libraries to handle cryptographic requirements. The PKCS #11 libraries are implemented according to the RSA Security Inc. PKCS #11 Cryptographic Token Interface (Cryptoki) standard.

FIGURE 1 Cryptographic Framework Levels


Oracle Solaris Utilities
IKE
OpenSSL ENGINE
Java JCE SunPKCS11
Kerberos
Consumer Interface (PKCS 11)
User-Level Cryptographic Framework
Provider Interface (PKCS 11)
Hardware Crypto Accelerators/ Keystores
Softtoken Keystore
Oracle Key Manager
TPM: Trusted Platform Module
Application Level
/dev/crypto
Kernel Level
IPsec
ZFS
Kerberos
SPARC T4+ Intel AESNI SSSE3, AVX Instructions
Consumer Interface
Kernel-Level Cryptographic Framework
Provider Interface
Kernel Algorithms (DES, 3DES, AES, Blowfish, RSA, MD5, SHA_1, ...)
Hardware Crypto Accelerators/ Keystores

At the kernel level, the framework currently handles cryptographic requirements for ZFS, Kerberos and IPsec, as well as hardware. User-level consumers include the OpenSSL engine, Java Cryptographic Extensions (JCE), `libsasl`, and IKE (Internet Key Protocol). The kernel SSL (`kssl`) proxy uses the Cryptographic Framework. For more information, see "SSL Kernel Proxy Encrypts Web Server Communications" in *Securing the Network in Oracle Solaris 11.3* and the `ksslcfg`(1M) man page.

Export law in the United States requires that the use of open cryptographic interfaces be licensed. The Cryptographic Framework satisfies the current law by requiring that kernel cryptographic providers and PKCS #11 cryptographic providers be signed. For further discussion, see the information about the `elfsign` command in "User-Level Commands in the Cryptographic Framework" on page 15.

The framework enables *providers* of cryptographic services to have their services used by many *consumers* in Oracle Solaris. Another name for providers is *plugins*. The framework supports three types of plugins:

- User-level plugins – Shared objects that provide services by using PKCS #11 libraries, such as `/var/user/$USER/pkcs11_softtoken.so.1`.
- Kernel-level plugins – Kernel modules that provide implementations of cryptographic algorithms in software, such as AES.

  Many of the algorithms in the framework are optimized for x86 with SSSE3 instructions and AVX instructions and for SPARC hardware. For T-Series optimizations, see "Cryptographic Framework Optimizations for SPARC Based Systems" on page 19.
- Hardware plugins – Device drivers and their associated hardware accelerators. The Niagara chips and Oracle's `ncp` and `n2cp` device drivers are one example. A hardware accelerator offloads expensive cryptographic functions from the operating system. Sun Crypto Accelerator 6000 board is one example.

The framework implements a standard interface, the PKCS #11, v2.20 amendment 3 library, for user-level providers. The library can be used by third-party applications to reach providers. Third parties can also add signed libraries, signed kernel algorithm modules, and signed device drivers to the framework. These plugins are added when the Image Packaging System (IPS) installs the third-party software. For a diagram of the major components of the framework, see Figure 1, "Cryptographic Framework Levels," on page 11.

Concepts in the Cryptographic Framework

Note the following descriptions of concepts and corresponding examples that are useful when working with the Cryptographic Framework.

- **Algorithms –** Cryptographic algorithms are established, recursive computational procedures that encrypt or hash input. Encryption algorithms can be symmetric or asymmetric. Symmetric algorithms use the same key for encryption and decryption. Asymmetric algorithms, which are used in public-key cryptography, require two keys. Hashing functions are also algorithms.

  Examples of algorithms include:
  - Symmetric algorithms, such as AES
  - Asymmetric algorithms, such as Diffie-Hellman and RSA
  - Hashing functions, such as SHA256
- **Consumers –** Users of the cryptographic services that come from providers. Consumers can be applications, end users, or kernel operations.

Examples of consumers include:

- Applications, such as IKE
- End users, such as a regular user who runs the `encrypt` command
- Kernel operations, such as IPsec

- **Keystore –** In the Cryptographic Framework, persistent storage for token objects, often used interchangeably with **token**. For information about a reserved keystore, see **Metaslot** in this list of definitions.
- **Mechanism –** The application of a mode of an algorithm for a particular purpose.

  For example, a DES mechanism that is applied to authentication, such as CKM_DES_MAC, is a separate mechanism from a DES mechanism that is applied to encryption, CKM_DES_CBC_PAD.
- **Metaslot –** A single slot that presents a union of the capabilities of other slots which are loaded in the framework. The metaslot eases the work of dealing with all of the capabilities of the providers that are available through the framework. When an application that uses the metaslot requests an operation, the metaslot determines which actual slot will perform the operation. Metaslot capabilities are configurable, but configuration is not required. The metaslot is on by default. For more information, see the `cryptoadm`(1M) man page.

  The metaslot does not have its own keystore. Rather, the metaslot reserves the use of a keystore from one of the actual slots in the Cryptographic Framework. By default, the metaslot reserves the `Sun Crypto Softtoken` keystore. The keystore that is used by the metaslot is not shown as one of the available slots.

  Users can specify an alternate keystore for metaslot by setting the environment variables `${METASLOT_OBJECTSTORE_SLOT}` and `${METASLOT_OBJECTSTORE_TOKEN}`, or by running the `cryptoadm` command. For more information, see the `libpkcs11`(3LIB), `pkcs11_softtoken`(5), and `cryptoadm`(1M) man pages.
- **Mode –** A version of a cryptographic algorithm. For example, CBC (Cipher Block Chaining) is a different mode from ECB (Electronic Code Book). The AES algorithm has modes such as CKM_AES_ECB and CKM_AES_CBC.
- **Policy –** The choice, by an administrator, of which mechanisms to make available for use. By default, all providers and all mechanisms are available for use. The enabling or disabling of any mechanism would be an application of policy. For examples of setting and applying policy, see "Administering the Cryptographic Framework" on page 36.
- **Providers –** Cryptographic services that consumers use. Providers plug in to the framework, and so are also called *plugins*.

  Examples of providers include:

  - PKCS #11 libraries, such as `/var/user/$USER/pkcs11_softtoken.so`
  - Modules of cryptographic algorithms, such as `aes` and `arcfour`
  - Device drivers and their associated hardware accelerators, such as the `mca` driver for the Sun Crypto Accelerator 6000

- **Slot –** An interface to one or more cryptographic devices. Each slot, which corresponds to a physical reader or other device interface, might contain a token. A token provides a logical view of a cryptographic device in the framework.
- **Token –** In a slot, a token provides a logical view of a cryptographic device in the framework.

Cryptographic Framework Commands and Plugins

The framework provides commands for administrators, for users, and for developers who supply providers.

- Administrative commands – The `cryptoadm` command provides a `list` subcommand to list the available providers and their capabilities. Regular users can run the `cryptoadm list` and the `cryptoadm --help` commands.

  All other `cryptoadm` subcommands require you to assume a role that includes the Crypto Management rights profile, or to become superuser. Subcommands such as `disable`, `install`, and `uninstall` are available for administering the framework. For more information, see the `cryptoadm`(1M) man page.

  The `svcadm` command is used to manage the `kcfd` daemon and to refresh cryptographic policy in the kernel. For more information, see the `svcadm`(1M) man page.
- User-level commands – The `digest` and `mac` commands provide file integrity services. The `encrypt` and `decrypt` commands protect files from eavesdropping. To use these commands, see Table 2, "Protecting Files With the Cryptographic Framework Task Map," on page 23.

Administrative Commands in the Cryptographic Framework

The `cryptoadm` command administers a running Cryptographic Framework. The command is part of the Crypto Management rights profile. This profile can be assigned to a role for secure administration of the Cryptographic Framework. You use the `cryptoadm` command to do the following:

- Disable or enable provider mechanisms
- Disable or enable the metaslot

You use the `svcadm` command to enable, refresh, and disable the cryptographic services daemon, `kcfd`. This command is part of the Service Management Facility (SMF) feature of Oracle Solaris. `svc:/system/cryptosvcs` is the service instance for the Cryptographic Framework. For more information, see the `smf`(5) and `svcadm`(1M) man pages.

User-Level Commands in the Cryptographic Framework

The Cryptographic Framework provides user-level commands to check the integrity of files, to encrypt files, and to decrypt files.

- `digest` command – Computes a message digest for one or more files or for stdin. A digest is useful for verifying the integrity of a file. SHA1 and MD5 are examples of digest functions.
- `mac` command – Computes a MAC for one or more files or for stdin. A MAC associates data with an authenticated message. A MAC enables a receiver to verify that the message came from the sender and that the message has not been tampered with. The `sha1_mac` and `md5_hmac` mechanisms can compute a MAC.
- `encrypt` command – Encrypts files or stdin with a symmetric cipher. The `encrypt -l` command lists the algorithms that are available. Mechanisms that are listed under a user-level library are available to the `encrypt` command. The framework provides AES, DES, 3DES (Triple-DES), and ARCFOUR mechanisms for user encryption.
- `decrypt` command – Decrypts files or stdin that were encrypted with the `encrypt` command. The `decrypt` command uses the identical key and mechanism that were used to encrypt the original file.
- `elfsign` command – Provides a means to sign providers to be used with the Cryptographic Framework. Typically, this command is run by the developer of a provider. The `elfsign` command has subcommands to request a certificate, sign binaries, and verify the signature on a binary. Unsigned binaries cannot be used by the Cryptographic Framework. Providers that have verifiable signed binaries can use the framework.

Plugins to the Cryptographic Framework

Third parties can plug their providers into the Cryptographic Framework. A third-party provider can be one of the following objects:

- PKCS #11 shared library
- Loadable kernel software module, such as an encryption algorithm, MAC function, or digest function
- Kernel device driver for a hardware accelerator

The objects from a provider must be signed with a certificate from Oracle. The certificate request is based on a private key that the third party selects, and a certificate that Oracle provides. The certificate request is sent to Oracle, which registers the third party and then issues the certificate. The third party then signs its provider object with the certificate from Oracle.

The loadable kernel software modules and the kernel device drivers for hardware accelerators must also register with the kernel. Registration is through the Cryptographic Framework SPI (service provider interface).

Cryptographic Framework and Zones

The global zone and each non-global zone has its own `system/cryptosvc` service, which manages the Cryptographic Framework. When the cryptographic service is enabled or refreshed in the global zone, the `kcfd` daemon starts in the global zone, user-level policy for the global zone is set, and kernel policy for the system is set. When the service is enabled or refreshed in a non-global zone, the `kcfd` daemon starts in the zone, and user-level policy for the zone is set. Kernel policy was set by the global zone.

For more information about zones, see *Introduction to Oracle Solaris Zones*. For more information about using SMF to manage persistent applications, see Chapter 1, "Introduction to the Service Management Facility" in *Managing System Services in Oracle Solaris 11.3* and the `smf`(5) man page.

Cryptographic Sources and FIPS 140-2

FIPS 140-2 is a U.S. Government computer security standard for cryptography modules.

Oracle Solaris systems offer two providers of cryptographic algorithms that are approved for FIPS 140-2 Level 1.

- The Cryptographic Framework of Oracle Solaris is a provider of two FIPS 140-2 approved modules. The *userland* module supplies cryptography for applications that run in user space. The *kernel* module provides cryptography for kernel-level processes.
- The OpenSSL object module provides FIPS 140-2 approved cryptography for SunSSH and web applications. For information about the implementations of Secure Shell in Oracle Solaris, see "About Secure Shell" in *Managing Secure Shell Access in Oracle Solaris 11.3*.

Note the following key considerations:

- Because FIPS 140-2 provider modules are CPU intensive, they are not enabled by default. As the system administrator, you are responsible for enabling the providers in FIPS 140-2 mode and configuring applications that use the FIPS 140-2 approved algorithms.

- If you have a strict requirement to use only FIPS 140-2 validated cryptography, you must be running the Oracle Solaris 11.3 SRU 5.6 release. Later Oracle Solaris releases build on this validated foundation and include software improvements that address performance, functionality, and reliability. Whenever possible, you should configure Oracle Solaris in FIPS 140-2 mode to take advantage of these improvements.

For more information, review the following:

- *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*
- "Creating a Boot Environment With FIPS 140-2 Enabled" on page 45

OpenSSL and Oracle Solaris

Oracle Solaris supports two implementations of OpenSSL:

- FIPS 140-2 capable OpenSSL
- Non-FIPS 140-2 capable OpenSSL

Both implementations are compatible with the latest OpenSSL version from the OpenSSL project. Use the `openssl version` command to determine the latest OpenSSL version. The libraries of both OpenSSL implementations are API/ABI compatible.

While both implementations are present in the operating system, only one implementation can be active at a time. To determine which OpenSSL implementation is active on the system, use the `pkg mediator openssl` command. For an example of installing the FIPS 140-2 capable OpenSSL and switching implementations, see "Example of Running in FIPS 140-2 Mode on an Oracle Solaris 11.3 SRU 5.6 System" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

♦ ♦ ♦ C H A P T E R 2

Cryptographic Optimizations and Hardware Acceleration on SPARC Based Systems

This chapter describes the optimization of cryptographic functions by the Cryptographic Framework, and the hardware acceleration of these optimized functions on current SPARC based systems. Affected systems include Oracle's SPARC T4, SPARC T5, SPARC M5, SPARC M6, SPARC T7, SPARC M7, and SPARC S7 Servers.

Cryptographic Framework Optimizations for SPARC Based Systems

The Cryptographic Framework supplies M-Series Servers beginning with SPARC M5, T-Series Servers beginning with SPARC T4, and SPARC S7 servers, with cryptographic mechanisms. Several mechanisms are optimized for these servers, including some symmetric key algorithms, asymmetric key algorithms, checksums, and digests.

Three cryptographic mechanisms are optimized for data at rest and in motion: `AES-CBC`, `AES-CFB128`, and `ARCFOUR`. The RSA and DSA cryptographic mechanisms are optimized for OpenSSL by optimizing arbitrary-precision arithmetic (bignum). Other optimizations include small packet performance for handshakes and data in motion.

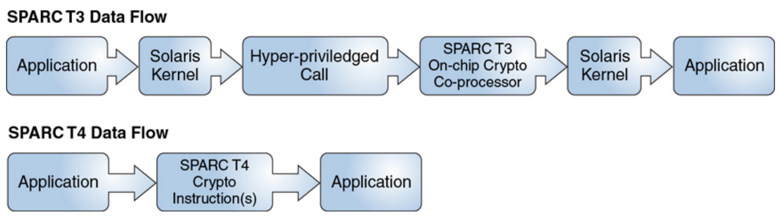
In turn, the SPARC based systems provide hardware acceleration of these cryptographic mechanisms to both the Cryptographic Framework and OpenSSL.

SPARC Acceleration of Optimized Cryptographic Functions

Beginning with the SPARC T4 microprocessor, new instructions to perform cryptographic functions are available directly in hardware. The instructions are non-privileged. Thus, any program can use the instructions. Because cryptography is performed directly on the hardware, cryptographic operations are faster than operations on legacy systems whose SPARC processors have separate processing units for cryptography.

The following comparison shows the differences in the data flow between SPARC T3 systems and SPARC T4 systems with cryptographic acceleration.

FIGURE 2 Data Flow Comparison Between SPARC T3 and SPARC T4 Servers



The following table provides a detailed comparison of cryptographic functions in SPARC T4 microprocessor units combined with specific Oracle Solaris releases.

TABLE 1 Cryptographic Performance on SPARC T4 and Later SPARC Based Systems

Feature/ Software Consumer	T3 and Previous Systems	T4 Systems Running Oracle Solaris 10	T4 Systems and Later Running Oracle Solaris 11
Secure Shell	Automatically enabled with Solaris 10 5/09 and later. Disable/Enable with the `UseOpenSSLEngine` option in `/etc/ssh/sshd_config`.	Requires patch 147707-01. Disable/enable with the `UseOpenSSLEngine` option in `/etc/ssh/sshd_config`.	Automatically enabled. For SunSSH, disable/enable with the `UseOpenSSLEngine` option in `/etc/ssh/sshd_config`.
Java/JCE	Automatically enabled. Configure in `$JAVA_HOME/jre/lib/security/java.security`	Automatically enabled. Configure in `$JAVA_HOME/jre/lib/security/java.security`	Automatically enabled. Configure in `$JAVA_HOME/jre/lib/security/java.security`
ZFS Crypto	Not available.	Not available.	HW crypto automatically enabled if dataset is encrypted.
IPsec	Automatically enabled.	Automatically enabled.	Automatically enabled.
OpenSSL	Use `-engine pkcs11` option.	Requires patch 151912-02 or newer. Use `-engine pkcs11` option.	The T4 optimization is automatically used.

Feature/ Software Consumer	T3 and Previous Systems	T4 Systems Running Oracle Solaris 10	T4 Systems and Later Running Oracle Solaris 11
			(Optionally use `-engine pkcs11`.) To use T4 crypto functions such as RSA or DSA, use this engine.
KSSL (Kernel SSL proxy)	Automatically enabled.	Automatically enabled.	Automatically enabled.
Oracle TDE	Not supported.	Pending patch.	Automatically enabled with Oracle DB 11.2.0.3 and ASO.
Apache SSL	Configure with `SSLCryptoDevice pkcs11`	Configure with `SSLCryptoDevice pkcs11`	Configure with `SSLCryptoDevice pkcs11`
Logical Domains	Assign crypto units to domains.	Functionality always available, no configuration required.	Functionality always available, no configuration required.

The T4 and later microprocessors provide on-chip encryption instruction accelerators with direct nonprivileged support for 15 industry-standard cryptographic algorithms: AES, Camellia, CRC32c, DES, 3DES, DH, DSA, ECC, MD5, RSA, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512.

For AES, the instructions include the following encryption and decryption optimizations:

- Key expansion – Expansions of the 128-bit, 192-bit, or 256-bit user-provided AES key into a key schedule. The schedule is used internally during encryption and decryption.
- Rounds or transformations – The number of rounds used (for example 10, 12, or 14) varies according to AES key length. By using larger keys, the user indicates a desire for more robust encryption at the cost of more computation.

These hardware cryptographic instructions are available and used automatically. The instructions are also embedded in the OpenSSL upstream code, so beginning with OpenSSL 1.0.1e, OpenSSL uses the instructions. To determine the version, run the `openssl version` command in a terminal window.

EXAMPLE 1 Determining Whether Your SPARC System Supports Cryptographic Optimizations

To determine whether the cryptographic optimizations are supported, use the `isainfo` command. The inclusion of `sparcv9` and `aes` in the output indicates that the system supports the optimizations.

```
$ isainfo -v
64-bit sparcv9 applications
        crc32c cbcond pause mont mpmul sha512 sha256 sha1 md5 camellia kasumi
        des aes ima hpc vis3 fmaf asi_blk_init vis2 vis popc
```

EXAMPLE 2 Determining Whether Your SPARC System Is Running Cryptographic Optimizations

To determine whether your system is running SPARC T4 microprocessor optimizations, check for the `aes_t4` instruction in the OpenSSL `libcrypto.so` library. If the following command does not generate output, then your system does not use the SPARC T4 microprocessor optimizations.

```
$ nm /lib/libcrypto.so.1.0.0 | grep aes_t4 | head -5
[1273]  |   1840096|        52|OBJT |LOCL |0    |20     |aes_t4_128_cbc
[1344]  |   1842800|        52|OBJT |LOCL |0    |20     |aes_t4_128_ccm
[1283]  |   1840408|        52|OBJT |LOCL |0    |20     |aes_t4_128_cfb
[1286]  |   1840512|        52|OBJT |LOCL |0    |20     |aes_t4_128_cfb1
[1289]  |   1840616|        52|OBJT |LOCL |0    |20     |aes_t4_128_cfb8
```

For more information, refer to the following articles.

- "SPARC T4 OpenSSL Engine" (`https://blogs.oracle.com/DanX/entry/sparc_t4_openssl_engine`)
- "How to tell if SPARC T4 crypto is being used?" (`https://blogs.oracle.com/DanX/entry/how_to_tell_if_sparc`)
- "Exciting Crypto Advances with the T4 processor and Oracle Solaris 11" (`http://bubbva.blogspot.com/2011/11/exciting-crypto-advances-with-t4.html`)
- "SPARC T4 Digest and Crypto Optimizations in Solaris 11.1" (`https://blogs.oracle.com/danx/sparc-t4-digest-and-crypto-optimizations-in-solaris-111`)
- Oracle SPARC T7-1 Data Sheet (`http://www.oracle.com/us/products/servers-storage/sparc-t7-1-server-ds-2687047.pdf`)

♦ ♦ ♦ CHAPTER 3

Using the Cryptographic Framework

This chapter describes how to use the Cryptographic Framework, and covers the following topics:

- "Protecting Files With the Cryptographic Framework" on page 23
- "Administering the Cryptographic Framework" on page 36

Protecting Files With the Cryptographic Framework

This section describes how to generate symmetric keys, how to create checksums for file integrity, and how to protect files from eavesdropping. System users can run the commands described in this section, and developers can write scripts that use them.

To configure your system in FIPS 140-2 mode, you must use FIPS 140-2 validated algorithms, modes, and key lengths. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

The Cryptographic Framework can help you protect your files. The following task map points to procedures for listing the available algorithms, and for protecting your files cryptographically.

TABLE 2 Protecting Files With the Cryptographic Framework Task Map

Task	Description	For Instructions
Generate a symmetric key.	Generates a key of user-specified length. Optionally, stores the key in a file, a PKCS #11 keystore, or an NSS keystore. For FIPS 140-2 approved mode, select a key type, mode, and key length that has been validated for FIPS 140-2. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.	"How to Generate a Symmetric Key by Using the `pktool` Command" on page 24

Task	Description	For Instructions
Provide a checksum that ensures the integrity of a file.	Verifies that the receiver's copy of a file is identical to the file that was sent.	"How to Compute a Digest of a File" on page 30
Protect a file with a message authentication code (MAC).	Verifies to the receiver of your message that you were the sender.	"How to Compute a MAC of a File" on page 31
Encrypt a file, and then decrypt the encrypted file.	Protects the content of files by encrypting the file. Provides the encryption parameters to decrypt the file.	"How to Encrypt and Decrypt a File" on page 33

▼ How to Generate a Symmetric Key by Using the `pktool` Command

Some applications require a symmetric key for encryption and decryption of communications. In this procedure, you create a symmetric key and store it.

If your site has a random number generator, you can use the generator to create a random number for the key. This procedure does not use your site's random number generator.

1. **(Optional) If you plan to use a keystore, create it.**

   - **To create and initialize a PKCS #11 keystore, see "How to Generate a Passphrase by Using the `pktool setpin` Command" on page 65.**

   - **To create and initialize an NSS database, see the sample command in Example 29, "Protecting a Keystore With a Passphrase," on page 67.**

2. **Generate a random number for use as a symmetric key.**

   For FIPS 140-2 approved algorithms, select a key length that has been validated for FIPS 140-2. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

   Use one of the following methods.

   - **Generate a key and store it in a file.**

     The advantage of a file-stored key is that you can extract the key from this file to use in an application's key file, such as the `/etc/inet/secret/ipseckeys` file or IPsec. The usage statement shows the arguments.

     ```
     $ pktool genkey keystore=file
     ```

```
...genkey keystore=file
outkey=key-fn
[ keytype=aes|arcfour|des|3des|generic ]
[ keylen=key-size (AES, ARCFOUR or GENERIC only)]
[ print=y|n ]
```

`outkey=`*key-fn*

The filename where the key is stored.

`keytype=`*specific-symmetric-algorithm*

For a symmetric key of any length, the value is `generic`. For a particular algorithm, specify `aes`, `arcfour`, `des`, or `3des`.

`keylen=`*size-in-bits*

The length of the key in bits. The number must be divisible by 8. Do *not* specify for `des` or `3des`.

`print=y`

Prints the key to the terminal window. By default, the value of `print` is `n`.

- **Generate a key and store it in a PKCS #11 keystore.**

The advantage of the PKCS #11 keystore is that you can retrieve the key by its label. This method is useful for keys that encrypt and decrypt files. You must complete Step 1 before using this method. The usage statement shows the arguments. The brackets around the keystore argument indicate that when the keystore argument is not specified, the key is stored in the PKCS #11 keystore.

```
$ pktool genkey keystore=pkcs11
...genkey [ keystore=pkcs11 ]
label=key-label
[ keytype=aes|arcfour|des|3des|generic ]
[ keylen=key-size (AES, ARCFOUR or GENERIC only)]
[ token=token[:manuf[:serial]]]
[ sensitive=y|n ]
[ extractable=y|n ]
[ print=y|n ]
```

`label=`*key-label*

A user-specified label for the key. The key can be retrieved from the keystore by its label.

`keytype=`*specific-symmetric-algorithm*

For a symmetric key of any length, the value is `generic`. For a particular algorithm, specify `aes`, `arcfour`, `des`, or `3des`.

`keylen=`*size-in-bits*

The length of the key in bits. The number must be divisible by 8. Do *not* specify for `des` or `3des`.

`token=`*token*

The token name. By default, it is `Sun Software PKCS#11 softtoken`.

`sensitive=n`

Specifies the sensitivity of the key. When the value is `y`, the key cannot be printed by using the `print=y` argument. By default, the value of `sensitive` is `n`.

`extractable=y`

Specifies that the key can be extracted from the keystore. Specify `n` to prevent the key from being extracted.

`print=y`

Prints the key to the terminal window. By default, the value of `print` is `n`.

- **Generate a key and store it in an NSS keystore.**

  You must complete Step 1 before using this method. The usage statement shows the arguments.

```
$ pktool genkey keystore=nss
...genkey keystore=nss
label=key-label
[ keytype=aes|arcfour|des|3des|generic ]
[ keylen=key-size (AES, ARCFOUR or GENERIC only)]
[ token=token[:manuf[:serial]]]
[ dir=directory-path ]
[ prefix=DBprefix ]
```

`label=`*key-label*

A user-specified label for the key. The key can be retrieved from the keystore by its label.

`keytype=`*specific-symmetric-algorithm*

For a symmetric key of any length, the value is `generic`. For a particular algorithm, specify `aes`, `arcfour`, `des`, or `3des`.

For FIPS 140-2 approved algorithms, select a key type that has been validated for FIPS 140-2. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

`keylen=`*size-in-bits*

The length of the key in bits. The number must be divisible by 8. Do *not* specify for `des` or `3des`.

For FIPS 140-2 approved algorithms, select a key length that has been validated for FIPS 140-2. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

`token=`*token*

The token name. By default, it is the NSS internal token.

`dir=`*directory*

The directory path to the NSS database. By default, *directory* is the current directory.

`prefix=`*directory*

The prefix to the NSS database. The default is no prefix.

3. **(Optional) Verify that the key exists.**

Use one of the following commands, depending on where you stored the key.

- **Verify the key in the *key-fn* file.**

```
$ pktool list keystore=file objtype=key [infile=key-fn]
Found n keys.
Key #1 - keytype:location (keylen)
```

- **Verify the key in the PKCS #11 or the NSS keystore.**

```
For PKCS #11, use the following command:

$ pktool list keystore=pkcs11 objtype=key
Enter PIN for keystore:
Found n keys.
Key #1 - keytype:location (keylen)
```

Alternately, replace `keystore=pkcs11` with `keystore=nss` in the command.

Example 3 Creating a Symmetric Key by Using the `pktool` Command

In the following example, a user creates a PKCS #11 keystore for the first time and then generates a large symmetric key for an application. Finally, the user verifies that the key is in the keystore.

Note that the initial password for a PKCS #11 keystore is `changeme`. The initial password for an NSS keystore is an empty password.

```
$ pktool setpin
Create new passphrase:xxxxxxxx
Re-enter new passphrase:xxxxxxxx
Passphrase changed.
$ pktool genkey label=specialappkey keytype=generic keylen=1024
Enter PIN for Sun Software PKCS#11 softtoken  :xxxxxxxx

$ pktool list objtype=key
Enter PIN for Sun Software PKCS#11 softtoken  :     Type password
No.      Key Type      Key Len.      Key Label
----------------------------------------------------
Symmetric keys:
1        Symmetric     1024          specialappkey
```

Example 4 Creating a FIPS 140-2 Approved AES Key by Using the `pktool` Command

In the following example, a secret key for the AES algorithm is created using a FIPS 140-2 approved algorithm and key length. The key is stored in a local file for later decryption. The command protects the file with `400` permissions. When the key is created, the `print=y` option displays the generated key in the terminal window.

The user who owns the keyfile retrieves the key by using the `od` command.

```
$ pktool genkey keystore=file outkey=256bit.file1 keytype=aes keylen=256 print=y
Key Value ="aaa2df1d10f02eaee2595d48964847757a6a49cf86c4339cd5205c24ac8c8873"
$ od -x 256bit.file1

0000000 aaa2 df1d 10f0 2eae e259 5d48 9648 4775
0000020 7a6a 49cf 86c4 339c d520 5c24 ac8c 8873
0000040
```

Example 5 Creating a Symmetric Key for IPsec Security Associations

In the following example, the administrator manually creates the keying material for IPsec SAs and stores them in files. Then, the administrator copies the keys to the `/etc/inet/secret/ipseckeys` file and destroys the original files.

First, the administrator creates and displays the keys that the IPsec policy requires:

```
$ pktool genkey keystore=file outkey=ipencrin1 keytype=generic keylen=192 print=y
Key Value ="294979e512cb8e79370dabecadc3fcbb849e78d2d6bd2049"
$ pktool genkey keystore=file outkey=ipencrout1 keytype=generic keylen=192 print=y
Key Value ="9678f80e33406c86e3d1686e50406bd0434819c20d09d204"
$ pktool genkey keystore=file outkey=ipspi1 keytype=generic keylen=32 print=y
Key Value ="acbeaa20"
$ pktool genkey keystore=file outkey=ipspi2 keytype=generic keylen=32 print=y
Key Value ="19174215"
$ pktool genkey keystore=file outkey=ipsha21 keytype=generic keylen=256 print=y
Key Value ="659c20f2d6c3f9570bcee93e96d95e2263aca4eeb3369f72c5c786af4177fe9e"
$ pktool genkey keystore=file outkey=ipsha22 keytype=generic keylen=256 print=y
Key Value ="b041975a0e1fce0503665c3966684d731fa3dbb12fcf87b0a837b2da5d82c810"
```

Then, the administrator creates the following `/etc/inet/secret/ipseckeys` file:

```
##   SPI values require a leading 0x.
##   Backslashes indicate command continuation.
##
## for outbound packets on this system
add esp spi 0xacbeaa20 \
src 192.0.2.12 dst 192.0.2.57 \
encr_alg aes auth_alg sha256  \
encrkey  294979e512cb8e79370dabecadc3fcbb849e78d2d6bd2049 \
authkey  659c20f2d6c3f9570bcee93e96d95e2263aca4eeb3369f72c5c786af4177fe9e
##
## for inbound packets
add esp spi 0x19174215 \
src 192.0.2.57 dst 192.0.2.12 \
encr_alg aes auth_alg sha256  \
encrkey 9678f80e33406c86e3d1686e50406bd0434819c20d09d204 \
authkey b041975a0e1fce0503665c3966684d731fa3dbb12fcf87b0a837b2da5d82c810
```

After verifying that the syntax of the `ipseckeys` file is valid, the administrator destroys the original key files.

```
$ ipseckey -c /etc/inet/secret/ipseckeys
$ rm ipencrin1 ipencrout1 ipspi1 ipspi2 ipsha21 ipsha22
```

The administrator copies the `ipseckeys` file to the communicating system by using the `ssh` command or another secure mechanism. On the communicating system, the protections are reversed. The first entry in the `ipseckeys` file protects inbound packets, and the second entry protects outbound packets. No keys are generated on the communicating system.

Next Steps To proceed with using the key to create a message authentication code (MAC) for a file, see "How to Compute a MAC of a File" on page 31.

▼ How to Compute a Digest of a File

When you compute a digest of a file, you can check to see that the file has not been tampered with by comparing digest outputs. A digest does not alter the original file.

1. **List the available digest algorithms.**

```
$ digest -l
md5
sha1
sha224
sha256
sha384
sha512
```

Note - Whenever possible, select a FIPS 140-2 approved algorithm. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

2. **Compute the digest of the file and save the digest listing.**

Provide an algorithm with the `digest` command.

```
$ digest -v -a algorithm input-file > digest-listing
```

`-v`	Displays the output in the following format: *algorithm* `(`*input-file*`) =` *digest*
`-a` *algorithm*	The algorithm to use to compute a digest of the file. Type the algorithm as the algorithm appears in the output of Step 1.

Note - Whenever possible, select a FIPS 140-2 approved algorithm. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

input-file	The input file for the `digest` command.
digest-listing	The output file for the `digest` command.

Example 6 Computing a Digest With the SHA1 Mechanism

In the following example, the `digest` command uses the SHA1 mechanism to provide a directory listing. The results are placed in a file.

```
$ digest -v -a sha1 docs/* > $HOME/digest.docs.legal.05.07
$ more ~/digest.docs.legal.05.07
sha1 (docs/legal1) = 1df50e8ad219e34f0b911e097b7b588e31f9b435
sha1 (docs/legal2) = 68efa5a636291bde8f33e046eb33508c94842c38
sha1 (docs/legal3) = 085d991238d61bd0cfa2946c183be8e32cccf6c9
sha1 (docs/legal4) = f3085eae7e2c8d008816564fdf28027d10e1d983
```

▼ How to Compute a MAC of a File

A message authentication code, or MAC, computes a digest for the file and uses a secret key to further protect the digest. A MAC does not alter the original file.

1. **List the available mechanisms.**

```
$ mac -l
Algorithm       Keysize:  Min   Max
-----------------------------------
des_mac                    64    64
sha1_hmac                   8   512
md5_hmac                    8   512
sha224_hmac                 8   512
sha256_hmac                 8   512
sha384_hmac                 8  1024
sha512_hmac                 8  1024
```

Note - Each supported algorithm is an alias to the most commonly used and least restricted version of a particular algorithm type. The preceding output shows available algorithm names and the keysize for each algorithm. Whenever possible, use a supported algorithm that matches a FIPS 140-2 approved algorithm with a FIPS 140-2 approved key length, listed at "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

2. **Generate a symmetric key of the appropriate length.**

   You can provide either a passphrase from which a key will be generated or you can provide a key.

   - If you provide a passphrase, you must store or remember the passphrase. If you store the passphrase online, the passphrase file should be readable only by you.

- If you provide a key, it must be the correct size for the mechanism. You can use the `pktool` command. For the procedure and some examples, see "How to Generate a Symmetric Key by Using the `pktool` Command" on page 24.

3. **Create a MAC for a file.**

   Provide a key and use a symmetric key algorithm with the `mac` command.

   ```
   $ mac [-v] -a algorithm [-k keyfile | -K key-label [-T token]] input-file
   ```

`-v`	Displays the output in the following format: *algorithm* (*input-file*) = *mac*
`-a` *algorithm*	The algorithm to use to compute the MAC. Type the algorithm as the algorithm appears in the output of the `mac -l` command.
`-k` *keyfile*	The file that contains a key of algorithm-specified length.
`-K` *key-label*	The label of a key in the PKCS #11 keystore.
`-T` *token*	The token name. By default, it is `Sun Software PKCS#11 softtoken`. It is used only when the `-K` *key-label* option is used.
input-file	The input file for the MAC.

Example 7 Computing a MAC With SHA1_HMAC and a Passphrase

In the following example, the email attachment is authenticated with the SHA1_HMAC mechanism and a key that is derived from a passphrase. The MAC listing is saved to a file. If the passphrase is stored in a file, the file should not be readable by anyone but the user.

```
$ mac -v -a sha1_hmac email.attach
Enter passphrase:      Type passphrase
sha1_hmac (email.attach) = 2b31536d3b3c0c6b25d653418db8e765e17fe07b
$ echo "sha1_hmac (email.attach) = 2b31536d3b3c0c6b25d653418db8e765e17fe07b" \
>> ~/sha1hmac.daily.05.12
```

Example 8 Computing a MAC With SHA1_HMAC and a Key File

In the following example, the directory manifest is authenticated with the SHA1_HMAC mechanism and a secret key. The results are placed in a file.

```
$ mac -v -a sha1_hmac \
-k $HOME/keyf/05.07.mack64 docs/* > $HOME/mac.docs.legal.05.07
```

```
$ more ~/mac.docs.legal.05.07
sha1_hmac (docs/legal1) = 9b31536d3b3c0c6b25d653418db8e765e17fe07a
sha1_hmac (docs/legal2) = 865af61a3002f8a457462a428cdb1a88c1b51ff5
sha1_hmac (docs/legal3) = 076c944cb2528536c9aebd3b9fbe367e07b61dc7
sha1_hmac (docs/legal4) = 7aede27602ef6e4454748cbd3821e0152e45beb4
```

Example 9 Computing a MAC With SHA1_HMAC and a Key Label

In the following example, the directory manifest is authenticated with the SHA1_HMAC mechanism and a secret key. The results are placed in the user's PKCS #11 keystore. The user initially created the keystore and the password to the keystore by using the `pktool setpin` command.

```
$ mac -a sha1_hmac -K legaldocs0507 docs/*
Enter pin for Sun Software PKCS#11 softtoken:     Type password
```

To retrieve the MAC from the keystore, the user uses the verbose option, and provides the key label and the name of the directory that was authenticated.

```
$ mac -v -a sha1_hmac -K legaldocs0507  docs/*
Enter pin for Sun Software PKCS#11 softtoken:     Type password
sha1_hmac (docs/legal1) = 9b31536d3b3c0c6b25d653418db8e765e17fe07a
sha1_hmac (docs/legal2) = 865af61a3002f8a457462a428cdb1a88c1b51ff5
sha1_hmac (docs/legal3) = 076c944cb2528536c9aebd3b9fbe367e07b61dc7
sha1_hmac (docs/legal4) = 7aede27602ef6e4454748cbd3821e0152e45beb4
```

▼ How to Encrypt and Decrypt a File

When you encrypt a file, the original file is not removed or changed. The output file is encrypted.

For solutions to common errors related to the `encrypt` command, see the section that follows the examples.

Note - When encrypting and decrypting files, try to use FIPS 140-2 approved algorithms with approved key lengths whenever possible. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*. Run the `encrypt -l` command to view available algorithms and their key lengths.

1. **Create a symmetric key of the appropriate length.**

You can provide either a passphrase from which a key will be generated or you can provide a key.

- If you provide a passphrase, you must store or remember the passphrase. If you store the passphrase online, the passphrase file should be readable only by you.
- If you provide a key, it must be the correct size for the mechanism. You can use the `pktool` command. For the procedure and some examples, see "How to Generate a Symmetric Key by Using the `pktool` Command" on page 24.

2. **Encrypt a file.**

   Provide a key and use a symmetric key algorithm with the `encrypt` command.

```
$ encrypt -a algorithm [-v] \
[-k keyfile | -K key-label [-T token]] [-i input-file] [-o output-file]
```

`-a` *algorithm*	The algorithm to use to encrypt the file. Type the algorithm as the algorithm appears in the output of the `encrypt -l` command. Whenever possible, select a FIPS 140-2 approved algorithm. See "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.
`-k` *keyfile*	The file that contains a key of algorithm-specified length. The key length for each algorithm is listed, in bits, in the output of the `encrypt -l` command.
`-K` *key-label*	The label of a key in the PKCS #11 keystore.
`-T` *token*	The token name. By default, it is `Sun Software PKCS#11 softtoken`. It is used only when the `-K` *key-label* option is used.
`-i` *input-file*	The input file that you want to encrypt. This file is left unchanged by the command.
`-o` *output-file*	The output file that is the encrypted form of the input file.

Example 10 Creating an AES Key for Encrypting Your Files

In the following example, a user creates and stores an AES key in an existing PKCS #11 keystore for use in encryption and decryption. The user can verify that the key exists and can use the key, but cannot view the key itself.

```
$ pktool genkey label=MyAESkeynumber1 keytype=aes keylen=256
Enter PIN for Sun Software PKCS#11 softtoken  :xxxxxxxx
```

```
$ pktool list objtype=key
Enter PIN for Sun Software PKCS#11 softtoken  :xxxxxxxx
No.      Key Type      Key Len.      Key Label
----------------------------------------------------
Symmetric keys:
1        AES           256           MyAESkeynumber1
```

To use the key to encrypt a file, the user retrieves the key by its label.

```
$ encrypt -a aes -K MyAESkeynumber1 -i encryptthisfile -o encryptedthisfile
```

To decrypt the `encryptedthisfile` file, the user retrieves the key by its label.

```
$ decrypt -a aes -K MyAESkeynumber1 -i encryptedthisfile -o sameasencryptthisfile
```

Example 11 Encrypting and Decrypting With AES and a Passphrase

In this example, a file is encrypted with the AES algorithm. The key is generated from the passphrase. If the passphrase is stored in a file, the file should not be readable by anyone but the user.

```
$ encrypt -a aes -i ticket.to.ride -o ~/enc/e.ticket.to.ride
Enter passphrase:    xxxxxxxx
Re-enter passphrase: xxxxxxxx
```

The input file, `ticket.to.ride`, still exists in its original form.

To decrypt the output file, the user uses the same passphrase and encryption mechanism that encrypted the file.

```
$ decrypt -a aes -i ~/enc/e.ticket.to.ride -o ~/d.ticket.to.ride
Enter passphrase: xxxxxxxx
```

Example 12 Encrypting and Decrypting With AES and a Key File

In this example, a file is encrypted with the AES algorithm. AES mechanisms use a key of 128 bits, or 16 bytes.

```
$ encrypt -a aes -k ~/keyf/05.07.aes16 \
-i ticket.to.ride -o ~/enc/e.ticket.to.ride
```

The input file, `ticket.to.ride`, still exists in its original form.

To decrypt the output file, the user uses the same key and encryption mechanism that encrypted the file.

```
$ decrypt -a aes -k ~/keyf/05.07.aes16  \
-i ~/enc/e.ticket.to.ride -o ~/d.ticket.to.ride
```

Troubleshooting The following messages indicate that the key that you provided to the `encrypt` command is not permitted by the algorithm that you are using.

- `encrypt: unable to create key for crypto operation: CKR_ATTRIBUTE_VALUE_INVALID`
- `encrypt: failed to initialize crypto operation: CKR_KEY_SIZE_RANGE`

If you pass a key that does not meet the requirements of the algorithm, you must supply a better key by using one of the following methods:

- Use a passphrase. The framework then provides a key that meets the requirements.
- Pass a key size that the algorithm accepts. For example, the DES algorithm requires a key of 64 bits. The 3DES algorithm requires a key of 192 bits.

Administering the Cryptographic Framework

This section describes how to administer the software providers and the hardware providers in the Cryptographic Framework. You can, for example, disable the implementation of an algorithm from one software provider. You can then force the system to use the algorithm from a different software provider.

Caution - Do not disable the default providers that are included with the Oracle Solaris operating system. In particular, the `pkcs11_softtoken` provider is a required part of Oracle Solaris and must not be disabled by using the `cryptoadm` command.

Some of the cryptographic algorithms may be hardware accelerated. Administrators can run the following command to view a list of cryptographic algorithms for their system and check the `HW` column in the output:

```
$ cryptoadm list -vm provider='/usr/lib/security/$ISA/pkcs11_softtoken.so'`
```

For more information, see the `pkcs11_softtoken(5)` man page.

Note - An important component of administering the Cryptographic Framework is to plan and implement your policy regarding FIPS 140-2, the U.S. Government computer security standard for cryptography modules.

If you have a strict requirement to use only FIPS 140-2 validated cryptography, you must be running the Oracle Solaris 11.3 SRU 5.6 release. Later Oracle Solaris releases build on this validated foundation and include software improvements that address performance, functionality, and reliability. Whenever possible, you should configure Oracle Solaris in FIPS 140-2 mode to take advantage of these improvements.

Review *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3* and plan an overall FIPS 140-2 policy for your systems.

The following task map points to procedures for administering software and hardware providers in the Cryptographic Framework.

TABLE 3 Administering the Cryptographic Framework Task Map

Task	Description	For Instructions
Plan the FIPS 140-2 policy for your systems.	Decide on your plan for enabling FIPS 140-2 approved providers and consumers and implement your plan.	*Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*
List the providers in the Cryptographic Framework.	Lists the algorithms, libraries, and hardware devices that are available for use in the Cryptographic Framework.	"Listing Available Providers" on page 38
Enable FIPS 140-2 mode.	Runs the Cryptographic Framework to a U.S. government standard for cryptography modules.	"How to Create a Boot Environment With FIPS 140-2 Enabled" on page 45
Add a software provider.	Adds a PKCS #11 library or a kernel module to the Cryptographic Framework. The provider must be signed.	"How to Add a Software Provider" on page 44
Prevent the use of a user-level mechanism.	Removes a software mechanism from use. The mechanism can be enabled again.	"How to Prevent the Use of a User-Level Mechanism" on page 47
Temporarily disable mechanisms from a kernel module.	Temporarily removes a mechanism from use. Usually used for testing.	"How to Prevent the Use of a Kernel Software Mechanism" on page 49
Uninstall a library.	Removes a user-level software provider from use.	Example 18, "Permanently Removing a User-Level Library," on page 49
Uninstall a kernel provider.	Removes a kernel software provider from use.	Example 20, "Temporarily Removing Kernel Software Provider Availability," on page 50
Disable mechanisms from a hardware provider.	Ensures that selected mechanisms on a hardware accelerator are not used.	"How to Disable Hardware Provider Mechanisms and Features" on page 52

Task	Description	For Instructions
Restart or refresh cryptographic services.	Ensures that cryptographic services are available.	"How to Refresh or Restart All Cryptographic Services" on page 54

Listing Available Providers

Hardware providers are automatically located and loaded. For more information, see the `driver.conf(4)` man page.

When you have hardware that expects to plug in to the Cryptographic Framework, the hardware registers with the SPI in the kernel. The framework checks that the hardware driver is signed. Specifically, the framework checks that the object file of the driver is signed with a certificate that Oracle issues.

For example, the Sun Crypto Accelerator 6000 board (`mca`) and the `/dev/crypto` driver for the T-Series systems plug hardware mechanisms into the framework.

For information about getting your provider signed, see the information about the `elfsign` command in "User-Level Commands in the Cryptographic Framework" on page 15.

To list available providers, you use the `cryptoadm list` commands with different options depending on the specific information you want to obtain.

- Listing all the providers on the system.

  The contents and format of the providers list varies for different Oracle Solaris releases and different hardware platforms. Run the `cryptoadm list` command on your system to see the providers that your system supports. Only those mechanisms at the user level are available for direct use by regular users.

  ```
  $ cryptoadm list
  User-level providers:      /* for applications */
  Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
  Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so
  Provider: /usr/lib/security/$ISA/pkcs11_tpm.so

  Kernel software providers:      /* for IPsec, kssl, Kerberos */
  des
  aes
  arcfour
  blowfish
  camellia
  ```

```
ecc
sha1
sha2
md4
md5
rsa
swrand
n2rng/0      /* for hardware */
ncp/0
n2cp/0
```

- Listing the providers and their mechanisms in the Cryptographic Framework.

  You can view the strength and modes, such as `ECB` and `CBC`, of the available mechanisms. However, some of the listed mechanisms might be unavailable for use. See the next item for instructions about how to list which mechanisms can be used.

  The following output is truncated for display purposes.

```
$ cryptoadm list -m [provider=provider]
User-level providers:
=====================

Provider: /usr/lib/security/$ISA/pkcs11_kernel.so

Mechanisms:
CKM_DSA
CKM_RSA_X_509
CKM_RSA_PKCS
...
CKM_SHA256_HMAC_GENERAL
CKM_SSL3_MD5_MAC

Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so
Mechanisms:
CKM_DES_CBC
CKM_DES_CBC_PAD
CKM_DES_ECB
CKM_DES_KEY_GEN
CKM_DES_MAC_GENERAL
...
CKM_ECDSA_SHA1
CKM_ECDH1_DERIVE

Provider: /usr/lib/security/$ISA/pkcs11_tpm.so
/usr/lib/security/$ISA/pkcs11_tpm.so: no slots presented.
```

```
Kernel providers:
==========================
des: CKM_DES_ECB,CKM_DES_CBC,CKM_DES3_ECB,CKM_DES3_CBC
aes: CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR,CKM_AES_CCM, \
     CKM_AES_GCM,CKM_AES_GMAC,
CKM_AES_CFB128,CKM_AES_XTS,CKM_AES_XCBC_MAC
arcfour: CKM_RC4
blowfish: CKM_BLOWFISH_ECB,CKM_BLOWFISH_CBC
ecc: CKM_EC_KEY_PAIR_GEN,CKM_ECDH1_DERIVE,CKM_ECDSA, \
     CKM_ECDSA_SHA1
sha1: CKM_SHA_1,CKM_SHA_1_HMAC,CKM_SHA_1_HMAC_GENERAL
sha2: CKM_SHA224,CKM_SHA224_HMAC,...CKM_SHA512_256_HMAC_GENERAL

md4: CKM_MD4
md5: CKM_MD5,CKM_MD5_HMAC,CKM_MD5_HMAC_GENERAL
rsa: CKM_RSA_PKCS,CKM_RSA_X_509,CKM_MD5_RSA_PKCS, \
     CKM_SHA1_RSA_PKCS,CKM_SHA224_RSA_PKCS,
CKM_SHA256_RSA_PKCS,CKM_SHA384_RSA_PKCS,CKM_SHA512_RSA_PKCS
swrand: No mechanisms presented.
n2rng/0: No mechanisms presented.
ncp/0: CKM_DSA,CKM_RSA_X_509,CKM_RSA_PKCS,CKM_RSA_PKCS_KEY_PAIR_GEN,
CKM_DH_PKCS_KEY_PAIR_GEN,CKM_DH_PKCS_DERIVE,CKM_EC_KEY_PAIR_GEN,
CKM_ECDH1_DERIVE,CKM_ECDSA
n2cp/0: CKM_DES_CBC,CKM_DES_CBC_PAD,CKM_DES_ECB,CKM_DES3_CBC, \
     ...CKM_SSL3_SHA1_MAC
```

- Listing the available cryptographic mechanisms.

  Policy determines which mechanisms are available for use. The administrator sets the policy. An administrator can choose to disable mechanisms from a particular provider. The `-p` option displays the list of mechanisms that are permitted by the policy that the administrator has set.

  ```
  $ cryptoadm list -p [provider=provider]
  User-level providers:
  =====================
  /usr/lib/security/$ISA/pkcs11_kernel.so: \
       all mechanisms are enabled.random is enabled.
  /usr/lib/security/$ISA/pkcs11_softtoken.so: \
       all mechanisms are enabled, random is enabled.
  /usr/lib/security/$ISA/pkcs11_tpm.so: all mechanisms are enabled.

  Kernel providers:
  ==========================
  ```

```
   des: all mechanisms are enabled.
   aes: all mechanisms are enabled.
   arcfour: all mechanisms are enabled.
   blowfish: all mechanisms are enabled.
   ecc: all mechanisms are enabled.
   sha1: all mechanisms are enabled.
   sha2: all mechanisms are enabled.
   md4: all mechanisms are enabled.
   md5: all mechanisms are enabled.
   rsa: all mechanisms are enabled.
   swrand: random is enabled.
   n2rng/0: all mechanisms are enabled. random is enabled.
   ncp/0: all mechanisms are enabled.
   n2cp/0: all mechanisms are enabled.
```

The following examples show additional specific uses of the `cryptoadm list` command.

EXAMPLE 13 Listing Cryptographic Information of a Specific Provider

Specifying the provider in the `cryptoadm` *options* command limits the output only to information that is applicable to the provider.

```
$ cryptoadm enable provider=dca/0 random
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled, except CKM_MD5, CKM_MD5_HMAC,...
random is enabled.
```

The following output shows only the mechanisms are enabled. The random generator continues to be disabled.

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled, except CKM_MD5,CKM_MD5_HMAC,....

$ cryptoadm enable provider=dca/0 mechanism=all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is disabled.
```

The following output shows every feature and mechanism on the board is enabled.

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms ar enabled, except CKM_DES_ECB,CKM_DES3_ECB.
random is disabled.

$ cryptoadm enable provider=dca/0 all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is enabled.
```

EXAMPLE 14 Finding User-Level Cryptographic Mechanisms Only

In the following example, all mechanisms that the user-level library, `pkcs11_softtoken`, offers are listed.

```
$ cryptoadm list -m provider=/usr/lib/security/\
   $ISA/pkcs11_softtoken.so
Mechanisms:
CKM_DES_CBC
CKM_DES_CBC_PAD
CKM_DES_ECB
CKM_DES_KEY_GEN
CKM_DES_MAC_GENERAL
CKM_DES_MAC
…
CKM_ECDSA
CKM_ECDSA_SHA1
CKM_ECDH1_DERIVE
```

EXAMPLE 15 Determining Which Cryptographic Mechanisms Perform Which Functions

Mechanisms perform specific cryptographic functions, such as signing or key generation. The `-v -m` options display every mechanism and its functions.

In this example, the administrator wants to determine for which functions the `CKM_ECDSA*` mechanisms can be used.

```
$ cryptoadm list -vm
User-level providers:
=====================
Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
Number of slots: 3
Slot #2
Description: ncp/0 Crypto Accel Asym 1.0
...
CKM_ECDSA                    163  571  X  .  .  .  X  .  X  .  .  .  .  .  .  .  .
...

Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so
...
CKM_ECDSA       112 571  .  .  .  .  X  .  X  .  .  .  .  .  .  .  .
CKM_ECDSA_SHA1  112 571  .  .  .  .  X  .  X  .  .  .  .  .  .  .  .
...
Kernel providers:
=================
...
ecc: CKM_EC_KEY_PAIR_GEN,CKM_ECDH1_DERIVE,CKM_ECDSA,CKM_ECDSA_SHA1
...
```

The listing indicates that these mechanisms are available from the following user-level providers:

- `CKM_ECDSA` and `CKM_ECDSA_SHA1` – As software implementation in `/usr/lib/security/$ISA/pkcs11_softtoken.so` library
- `CKM_ECDSA` – Accelerated by `ncp/0 Crypto Accel Asym 1.0` in `/usr/lib/security/$ISA/pkcs11_kernel.so` library

Each item in an entry represents a piece of information about the mechanism. For these ECC mechanisms, the listing indicates the following:

- Minimum length – 112 bytes
- Maximum length – 571 bytes
- Hardware – Is or is not available on hardware.
- Encrypt – Is not used to encrypt data.
- Decrypt – Is not used to decrypt data.
- Digest – Is not used to create message digests.
- Sign – Is used to sign data.
- Sign + Recover – Is not used to sign data, where the data can be recovered from the signature.
- Verify – Is used to verify signed data.
- Verify + Recover– Is not used to verify data that can be recovered from the signature.
- Key generation – Is not used to generate a private key.
- Pair generation – Is not used to generate a key pair.
- Wrap – Is not used to wrap. that is, encrypt, an existing key.
- Unwrap – Is not used to unwrap a wrapped key.
- Derive – Is not used to derive a new key from a base key.
- EC Caps – Absent EC capabilities that are not covered by previous items

Adding a Software Provider

The following procedure explains how to add providers to the system. You must become an administrator who is assigned the Crypto Management rights profile. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

▼ How to Add a Software Provider

1. **List the software providers that are available to the system.**

   ```
   $ cryptoadm list
   User-level providers:
   Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
   Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so

   Kernel software providers:
   des
   aes
   arcfour
   blowfish
   camellia
   ecc
   sha1
   sha2
   md5
   rsa
   swrand
   n2rng/0
   ```

2. **Add the package for a new provider from a repository.**

   In this example, the `pkcs11_tpm` provider is added.

   ```
   $ pkg install system/library/security/pkcs11_tpm
   ```

3. **Register the new provider with the Cryptographic Framework.**

   ```
   $ cryptoadm install provider='/usr/lib/security/$ISA/pkcs11_tpm.so'
   ```

4. **Locate the new provider on the list.**

   In this case, a new user-level software provider was installed.

   ```
   $ cryptoadm list

   User-level providers:
   Provider: /usr/lib/security/$ISA/pkcs11_kernel.so
   Provider: /usr/lib/security/$ISA/pkcs11_softtoken.so
   Provider: /usr/lib/security/$ISA/pkcs11_tpm.so      <  added provider

   Kernel providers:
   des
   aes
   arcfour
   blowfish
   ```

```
camellia
ecc
sha1
sha2
md5
rsa
swrand
n2rng/0
```

Creating a Boot Environment With FIPS 140-2 Enabled

By default, FIPS 140-2 mode is disabled in Oracle Solaris. In this procedure, you create a new boot environment (BE) for FIPS 140-2 mode, then enable FIPS 140-2 and boot into the new BE.

Caution - A FIPS 140-2 enabled system runs compliance tests that can cause a panic if they fail. Therefore, you need a BE to boot into while you debug issues with the FIPS 140-2 boundary.

For an overview of FIPS 140-2, see *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*. See also "Cryptographic Sources and FIPS 140-2" on page 16 and the `cryptoadm`(1M) man page.

How to Create a Boot Environment With FIPS 140-2 Enabled

Before You Begin You must assume the `root` role. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

1. **Determine whether the system is in FIPS 140-2 mode.**

```
$ cryptoadm list fips-140
User-level providers:
=====================
/usr/lib/security/$ISA/pkcs11_softtoken: FIPS 140 mode is disabled.

Kernel software providers:
==========================
des: FIPS 140 mode is disabled.
aes: FIPS 140 mode is disabled.
```

```
ecc: FIPS 140 mode is disabled.
sha1: FIPS 140 mode is disabled.
sha2: FIPS 140 mode is disabled.
rsa: FIPS 140 mode is disabled.
swrand: FIPS 140 mode is disabled.

Kernel hardware providers:
=========================:
```

2. **Create a new BE for your FIPS 140-2 version of the Cryptographic Framework.**

   Before you enable FIPS 140-2 mode, you must first create, activate, and boot a new BE by using the `beadm` command.

   a. **Create a BE based on your current BE.**

      In this example, you create a BE named `S11.3-FIPS`.

      ```
      # beadm create S11.3-FIPS-140
      ```

   b. **Activate that BE.**

      ```
      # beadm activate S11.3-FIPS-140
      ```

   c. **Reboot the system.**

   d. **Enable FIPS 140-2 mode in the new BE.**

      If the `fips-140` package is not yet loaded, this command also loads the package.

      ```
      # cryptoadm enable fips-140
      ```

Note - This subcommand does not disable the non-FIPS 140-2 approved algorithms from the user-level `pkcs11_softtoken` library and the kernel software providers. The consumers of the framework are responsible for using only FIPS 140-2 approved algorithms.

For more information about the effects of FIPS 140-2 mode, see *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3* and the `cryptoadm`(1M) man page.

3. **(Optional) To run without FIPS 140-2 enabled, disable FIPS 140-2 mode.**

   You can reboot the original BE or disable FIPS 140-2 in the current BE.

   - **Boot the original BE.**

     ```
     $ beadm list
     BE                Active Mountpoint Space   Policy Created
     ```

```
--                ------ ---------- -----   ------ -------
S11.3             -      -          48.22G   static 2012-10-10 10:10
S11.3-FIPS-140    NR     /          287.01M  static 2012-11-18 18:18
# beadm activate S11.1
# beadm list
BE                Active Mountpoint Space   Policy Created
--                ------ ---------- -----   ------ -------
S11.3             R      -          48.22G   static 2012-10-10 10:10
S11.3-FIPS-140    N      /          287.01M  static 2012-11-18 18:18
# reboot
```

- **Disable FIPS 140-2 mode in the current BE and reboot.**

```
# cryptoadm disable fips-140
```

Note - FIPS 140-2 mode remains in operation until the system is rebooted.

```
# reboot
```

Preventing the Use of Mechanisms

If some of the cryptographic mechanisms from a library provider should not be used, you can remove selected mechanisms. You might consider preventing the use of mechanisms if, for example, the same mechanism in another library performs better, or if a security vulnerability is being investigated.

If the Cryptographic Framework provides multiple modes of a provider such as AES, you might remove a slow mechanism from use, or a corrupted mechanism. You might also use this procedure to remove an algorithm with proven security vulnerabilities.

You can selectively disable mechanisms and the random number feature from a hardware provider. To enable them again, see Example 23, "Enabling Mechanisms and Features on a Hardware Provider," on page 53. The hardware in this example provides a random number generator.

▼ How to Prevent the Use of a User-Level Mechanism

Before You Begin You must become an administrator who is assigned the Crypto Management rights profile. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

1. **List the mechanisms that are offered by a particular user-level software provider.**

```
$ cryptoadm list -m provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so:
CKM_DES_CBC,CKM_DES_CBC_PAD,CKM_DES_ECB,CKM_DES_KEY_GEN,
CKM_DES3_CBC,CKM_DES3_CBC_PAD,CKM_DES3_ECB,CKM_DES3_KEY_GEN,
CKM_AES_CBC,CKM_AES_CBC_PAD,CKM_AES_ECB,CKM_AES_KEY_GEN,
...
```

2. **List the mechanisms that are available for use.**

```
$ cryptoadm list -p
user-level providers:
=====================
...
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled.
random is enabled.
...
```

3. **Disable the mechanisms that should not be used.**

```
$ cryptoadm disable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so \
> mechanism=CKM_DES_CBC,CKM_DES_CBC_PAD,CKM_DES_ECB
```

4. **List the mechanisms that are available for use.**

```
$ cryptoadm list -p provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled,
except CKM_DES_ECB,CKM_DES_CBC_PAD,CKM_DES_CBC. random is enabled.
```

Example 16 Enabling a User-Level Software Provider Mechanism

In this example, a disabled AES mechanism is again made available for use.

```
$ cryptoadm list -m provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so:
CKM_DES_CBC,CKM_DES_CBC_PAD,CKM_DES_ECB,CKM_DES_KEY_GEN,
CKM_DES3_CBC,CKM_DES3_CBC_PAD,CKM_DES3_ECB,CKM_DES3_KEY_GEN,CKM_AES_ECB
…
$ cryptoadm list -p provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled,
except CKM_AES_ECB,CKM_DES_CBC_PAD,CKM_DES_CBC. random is enabled.
$ cryptoadm enable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so \
> mechanism=CKM_AES_ECB
$ cryptoadm list -p provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled,
except CKM_DES_CBC_PAD,CKM_DES_CBC. random is enabled.
```

Example 17 Enabling All User-Level Software Provider Mechanisms

In the following example, all mechanisms from the user-level library are enabled.

```
$ cryptoadm enable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so all
$ cryptoadm list -p provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled.
random is enabled.
```

Example 18 Permanently Removing a User-Level Library

In this example, a `libpkcs11.so.1` library from the `/opt` directory is removed.

```
$ cryptoadm uninstall provider=/opt/lib/\$ISA/libpkcs11.so.1
$ cryptoadm list
user-level providers:
/usr/lib/security/$ISA/pkcs11_kernel.so
/usr/lib/security/$ISA/pkcs11_softtoken.so
/usr/lib/security/$ISA/pkcs11_tpm.so

kernel providers:
...
```

▼ How to Prevent the Use of a Kernel Software Mechanism

Before You Begin You must become an administrator who is assigned the Crypto Management rights profile. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

1. **List the mechanisms that are offered by a particular kernel software provider.**

```
$ cryptoadm list -m provider=aes
aes: CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR,CKM_AES_CCM,CKM_AES_GCM,
CKM_AES_GMAC,CKM_AES_CFB128,CKM_AES_XTS,CKM_AES_XCBC_MAC
```

2. **List the mechanisms that are available for use.**

```
$ cryptoadm list -p provider=aes
aes: all mechanisms are enabled.
```

3. **Disable the mechanism that should not be used.**

```
$ cryptoadm disable provider=aes mechanism=CKM_AES_ECB
```

4. **List the mechanisms that are available for use.**

```
$ cryptoadm list -p provider=aes
aes: all mechanisms are enabled, except CKM_AES_ECB.
```

Example 19 Enabling a Kernel Software Provider Mechanism

In this example, a disabled AES mechanism is again made available for use.

```
cryptoadm list -m provider=aes
aes: CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR,CKM_AES_CCM,
CKM_AES_GCM,CKM_AES_GMAC,CKM_AES_CFB128,CKM_AES_XTS,CKM_AES_XCBC_MAC
$ cryptoadm list -p provider=aes
aes: all mechanisms are enabled, except CKM_AES_ECB.
$ cryptoadm enable provider=aes mechanism=CKM_AES_ECB
$ cryptoadm list -p provider=aes
aes: all mechanisms are enabled.
```

Example 20 Temporarily Removing Kernel Software Provider Availability

In the following example, the AES provider is temporarily removed from use. The `unload` subcommand is useful to prevent a provider from being loaded automatically while the provider is being uninstalled. For example, the `unload` subcommand might be used when modifying a mechanism of this provider.

```
$ cryptoadm unload provider=aes

$ cryptoadm list
...
Kernel software providers:
des
aes (inactive)
arcfour
blowfish
ecc
sha1
sha2
md4
md5
rsa
swrand
n2rng/0
ncp/0
n2cp/0
```

The AES provider is unavailable until the Cryptographic Framework is refreshed.

```
$ svcadm refresh system/cryptosvc
```

```
$ cryptoadm list
...
Kernel software providers:
des
aes
arcfour
blowfish
camellia
ecc
sha1
sha2
md4
md5
rsa
swrand
n2rng/0
ncp/0
n2cp/0
```

If a kernel consumer is using the kernel software provider, the software is not unloaded. An error message is displayed and the provider continues to be available for use.

Example 21 Permanently Removing Software Provider Availability

In the following example, the AES provider is removed from use. Once removed, the AES provider does not appear in the policy listing of kernel software providers.

```
$ cryptoadm uninstall provider=aes
```

```
$ cryptoadm list
...
Kernel software providers:
des
arcfour
blowfish
camellia
ecc
sha1
sha2
md4
md5
rsa
swrand
n2rng/0
ncp/0
n2cp/0
```

Example 22 Reinstalling a Removed Kernel Software Provider

In the following example, the AES kernel software provider is reinstalled. To reinstall a removed kernel provider, you must enumerate the mechanisms to be installed.

```
$ cryptoadm install provider=aes \
mechanism=CKM_AES_ECB,CKM_AES_CBC,CKM_AES_CTR,CKM_AES_CCM,
CKM_AES_GCM,CKM_AES_GMAC,CKM_AES_CFB128,CKM_AES_XTS,CKM_AES_XCBC_MAC
```

```
$ cryptoadm list
...
Kernel software providers:
des
aes
arcfour
blowfish
camellia
ecc
sha1
sha2
md4
md5
rsa
swrand
n2rng/0
ncp/0
n2cp/0
```

▼ How to Disable Hardware Provider Mechanisms and Features

Before You Begin You must become an administrator who is assigned the Crypto Management rights profile. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

● **Choose the mechanisms or feature to disable.**

List the hardware provider.

```
$ cryptoadm list
...
Kernel hardware providers:
dca/0
```

- **Disable selected mechanisms.**

```
$ cryptoadm list -m provider=dca/0
dca/0: CKM_RSA_PKCS, CKM_RSA_X_509, CKM_DSA, CKM_DES_CBC, CKM_DES3_CBC
```

```
random is enabled.
$ cryptoadm disable provider=dca/0 mechanism=CKM_DES_CBC,CKM_DES3_CBC
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled except CKM_DES_CBC,CKM_DES3_CBC.
random is enabled.
```

- **Disable the random number generator.**

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is enabled.
$ cryptoadm disable provider=dca/0 random
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is disabled.
```

- **Disable all mechanisms. Do not disable the random number generator.**

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is enabled.
$ cryptoadm disable provider=dca/0 mechanism=all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are disabled. random is enabled.
```

- **Disable every feature and mechanism on the hardware.**

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is enabled.
$ cryptoadm disable provider=dca/0 all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are disabled. random is disabled.
```

Example 23 Enabling Mechanisms and Features on a Hardware Provider

In the following examples, disabled mechanisms on a piece of hardware are selectively enabled.

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled except CKM_RSA_PKCS,CKM_DES_ECB,CKM_DES3_ECB

. . .
random is enabled.
$ cryptoadm enable provider=dca/0 mechanism=CKM_RSA_PKCS
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled except CKM_DES_ECB,CKM_DES3_ECB.
random is enabled.
```

In the following example, only the random generator is enabled.

```
$ cryptoadm list -p provider=dca/0
```

```
dca/0: all mechanisms are enabled, except CKM_MD5,CKM_MD5_HMAC,….
random is disabled.
$ cryptoadm enable provider=dca/0 random
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled, except CKM_MD5,CKM_MD5_HMAC,….
random is enabled.
```

In the following example, only the mechanisms are enabled. The random generator continues to be disabled.

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled, except CKM_RSA_PKCS,CKM_RSA_X_509,….
random is disabled.
$ cryptoadm enable provider=dca/0 mechanism=all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is disabled.
```

In the following example, every feature and mechanism on the board is enabled.

```
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled, except CKM_RSA_PKCS,CKM_RSA_X_509.
random is disabled.
$ cryptoadm enable provider=dca/0 all
$ cryptoadm list -p provider=dca/0
dca/0: all mechanisms are enabled. random is enabled.
```

Refreshing or Restarting All Cryptographic Services

By default, the Cryptographic Framework is enabled. When the `kcfd` daemon fails for any reason, the Service Management Facility (SMF) can be used to restart cryptographic services. For more information, see the `smf`(5) and `svcadm`(1M) man pages. For the effect on zones of restarting cryptographic services, see "Cryptographic Framework and Zones" on page 16.

▼ How to Refresh or Restart All Cryptographic Services

Before You Begin You must become an administrator who is assigned the Crypto Management rights profile. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

1. **Check the status of cryptographic services.**

```
$ svcs cryptosvc
```

```
STATE          STIME    FMRI
offline         Dec_09   svc:/system/cryptosvc:default
```

2. **Enable cryptographic services.**

```
$ svcadm enable svc:/system/cryptosvc
```

Example 24 Refreshing Cryptographic Services

In the following example, cryptographic services are refreshed in the global zone. Therefore, kernel-level cryptographic policy in every non-global zone is also refreshed.

```
$ svcadm refresh system/cryptosvc
```

♦ ♦ ♦ CHAPTER 4

Managing Certificates in Oracle Solaris

The Key Management Framework (KMF) feature of Oracle Solaris provides tools and programming interfaces for managing public key objects. Public key objects include X.509 certificates and public/private key pairs. The formats for storing these objects can vary. KMF also provides a tool for managing policies that define the use of X.509 certificates by applications. KMF supports third-party plugins. Key management can require manual intervention, such as adding CA certificates to the directory where Oracle Solaris stores them.

This chapter covers the following topics:

- "Managing Public Key Technologies With the Key Management Framework" on page 57
- "Adding CA Certificates to the Oracle Solaris CA Keystore" on page 74

Managing Public Key Technologies With the Key Management Framework

KMF centralizes the management of public key technologies (PKI). Oracle Solaris has several different applications that make use of PKI technologies. Each application provides its own programming interfaces, key storage mechanisms, and administrative utilities. If an application provides a policy enforcement mechanism, the mechanism applies to that application only. With KMF, applications use a unified set of administrative tools, a single set of programming interfaces, and a single policy enforcement mechanism. These features manage the PKI needs of all applications that adopt these interfaces.

KMF unifies the management of public key technologies with the following interfaces:

- `pktool` command – Manages PKI objects, such as certificates, in a variety of keystores.
- `kmfcfg` command – Manages the PKI policy database and third-party plugins.

  PKI policy decisions include operations such as the validation method for an operation. Also, PKI policy can limit the scope of a certificate. For example, PKI policy might assert that a certificate can be used only for specific purposes. Such a policy would prevent that certificate from being used for other requests.

- KMF library – Contains programming interfaces that abstract the underlying keystore mechanism.

  Applications do not have to choose one particular keystore mechanism, but can migrate from one mechanism to another mechanism. The supported keystores are PKCS #11, NSS, and OpenSSL. The library includes a pluggable framework so that new keystore mechanisms can be added. Therefore, applications that use the new mechanisms would require only minor modifications to use a new keystore.

Key Management Framework Utilities

KMF provides methods for managing the storage of keys and provides the overall policy for the use of those keys. KMF can manage the policy, keys, and certificates for three public key technologies:

- Tokens from PKCS #11 providers, that is, from the Cryptographic Framework
- NSS, that is, Network Security Services
- OpenSSL, a file-based keystore

The `kmfcfg` tool can create, modify, or delete KMF policy entries. The tool also manages plugins to the framework. KMF manages keystores through the `pktool` command. For more information, see the `kmfcfg`(1) and `pktool`(1) man pages, and the following sections.

KMF Policy Management

KMF policy is stored in a database. This policy database is accessed internally by all applications that use the KMF programming interfaces. The database can constrain the use of the keys and certificates that are managed by the KMF library. When an application attempts to verify a certificate, the application checks the policy database. The `kmfcfg` command modifies the policy database.

KMF Plugin Management

The `kmfcfg` command provides the following subcommands for plugins:

- `list` *plugin* – Lists plugins that are managed by KMF.

- `install` *plugin* – Installs the plugin by the module's path name and creates a keystore for the plugin. To remove the plugin from KMF, you remove the keystore.
- `uninstall` *plugin* – Removes the plugin from KMF by removing its keystore.
- `modify` *plugin* – Enables the plugin to be run with an option that is defined in the code for the plugin, such as `debug`.

For more information, see the `kmfcfg`(1) man page. For the procedure, see "How to Manage Third-Party Plugins in KMF" on page 73.

KMF Keystore Management

KMF manages the keystores for three public key technologies, PKCS #11 tokens, NSS, and OpenSSL. For all of these technologies, the `pktool` command enables you to do the following:

- Generate a self-signed certificate
- Generate a certificate request
- Generate a symmetric key
- Generate a public/private key pair
- Generate a PKCS #10 certificate signing request (CSR) to be sent to an external certificate authority (CA) to be signed
- Sign a PKCS #10 CSR
- Import objects into the keystore
- List the objects in the keystore
- Delete objects from the keystore
- Download a CRL

For the PKCS #11 and NSS technologies, the `pktool` command also enables you to set a PIN by generating a passphrase for the keystore or for an object in the keystore.

For examples of using the `pktool` utility, see the `pktool`(1) man page and Table 4, "Using the Key Management Framework Task Map," on page 60.

Using the Key Management Framework

This section describes how to use the `pktool` command to manage your public key objects, such as passwords, passphrases, files, keystores, certificates, and CRLs.

The Key Management Framework (KMF) enables you to centrally manage public key technologies.

TABLE 4 Using the Key Management Framework Task Map

Task	Description	For Instructions
Create a certificate.	Creates a certificate for use by PKCS #11, NSS, or OpenSSL.	"How to Create a Certificate by Using the `pktool gencert` Command" on page 60
Export a certificate.	Creates a file with the certificate and its supporting keys. The file can be protected with a password.	"How to Export a Certificate and Private Key in PKCS #12 Format" on page 64
Import a certificate.	Imports a certificate from another system.	"How to Import a Certificate Into Your Keystore" on page 62
	Imports a certificate in PKCS #12 format from another system.	Example 26, "Importing a PKCS #12 File Into Your Keystore," on page 63
Generate a passphrase.	Generates a passphrase for access to a PKCS #11 keystore or an NSS keystore.	"How to Generate a Passphrase by Using the `pktool setpin` Command" on page 65
Generate a symmetric key.	Generates symmetric keys for use in encrypting files, in creating a MAC of a file, and for applications.	"How to Generate a Symmetric Key by Using the `pktool` Command" on page 24
Generate a key pair.	Generates a public/private key pair for use with applications.	"How to Generate a Key Pair by Using the `pktool genkeypair` Command" on page 67
Generate a PKCS #10 CSR.	Generates a PKCS #10 certificate signing request (CSR) for an external certificate authority (CA) to sign.	`pktool(1)` man page
Sign a PKCS #10 CSR.	Signs a PKCS #10 CSR.	"How to Sign a Certificate Request by Using the `pktool signcsr` Command" on page 71
Add a plugin to KMF.	Installs, modifies, and lists a plugin. Also, removes the plugin from the KMF.	"How to Manage Third-Party Plugins in KMF" on page 73

▼ How to Create a Certificate by Using the `pktool gencert` Command

This procedure creates a self-signed certificate and stores the certificate in the PKCS #11 keystore. As a part of this operation, an RSA public/private key pair is also created. The private key is stored in the keystore with the certificate.

1. **Generate a self-signed certificate.**

```
$ pktool gencert [keystore=keystore] label=label-name \
subject=subject-DN serial=hex-serial-number keytype=rsa/dsa keylen=key-size
```

`keystore=`*keystore*	Specifies the keystore by type of public key object. The value can be `nss`, `pkcs11`, or `file`. This keyword is optional.
`label=`*label-name*	Specifies a unique name that the issuer gives to the certificate.
`subject=`*subject-DN*	Specifies the distinguished name for the certificate.
`serial=`*hex-serial-number*	Specifies the serial number in hexadecimal format. The issuer of the certificate chooses the number, such as `0x0102030405`.
`keytype=`*key type*	Optional variable that specifies the type of private key associated with the certificate. Check the `pktool(1)` man page to find available key types for the selected keystore. To use a FIPS 140-2 approved key, check the approved key types at "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.
`keylen=`*key size*	Optional variable that specifies the length of the private key associated with the certificate. To use a FIPS 140-2 approved key, check the approved key lengths for the key type that you selected at "FIPS 140-2 Algorithms in the Cryptographic Framework" in *Using a FIPS 140-2 Enabled System in Oracle Solaris 11.3*.

2. **Verify the contents of the keystore.**

```
$ pktool list
Found number certificates.
1. (X.509 certificate)
Label:   label-name
ID: fingerprint that binds certificate to private key
Subject: subject-DN
Issuer:  distinguished-name
Serial:  hex-serial-number
n. ...
```

This command lists all certificates in the keystore. In the following example, the keystore contains one certificate only.

Example 25 Creating a Self-Signed Certificate by Using `pktool`

In the following example, a user at `My Company` creates a self-signed certificate and stores the certificate in a keystore for PKCS #11 objects. The keystore is initially empty. If the keystore has not been initialized, the PIN for the softtoken is `changeme`, and you can use the `pktool setpin` command to reset the PIN. Note that a FIPS 140-2 approved key type and key length, RSA 2048, is specified in the command options.

```
$ pktool gencert keystore=pkcs11 label="My Cert" \
subject="C=US, O=My Company, OU=Security Engineering Group, CN=MyCA" \
serial=0x000000001 keytype=rsa keylen=2048
Enter pin for Sun Software PKCS#11 softtoken:     Type PIN for token

$ pktool list
No.  Key Type  Key Len.  Key Label
----------------------------------------------------
Asymmetric public keys:
1    RSA                 My Cert
Certificates:
1    X.509 certificate
Label: My Cert
ID: d2:7e:20:04:a5:66:e6:31:90:d8:53:28:bc:ef:55:55:dc:a3:69:93
Subject: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
Issuer: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
...
...
Serial: 0x00000010
...
```

▼ How to Import a Certificate Into Your Keystore

This procedure describes how to import a file with PKI information that is encoded with PEM or with raw DER into your keystore. For an export procedure, see Example 28, "Exporting a Certificate and Private Key in PKCS #12 Format," on page 65.

1. **Import the certificate.**

```
$ pktool import keystore=keystore infile=infile-name label=label-name
```

2. **If you are importing certificates and private keys in PKCS #12 format, provide passwords when prompted.**

   a. **At the prompt, type the password for the file.**

If you are importing PKI information that is private, such as an export file in PKCS #12 format, the file requires a password. The creator of the file that you are importing provides you with the PKCS #12 password.

```
Enter password to use for accessing the PKCS12 file:    Type PKCS #12 password
```

b. **At the prompt, type the password for your keystore.**

```
Enter pin for Sun Software PKCS#11 softtoken:    Type PIN for token
```

3. **Verify the contents of the keystore.**

```
$ pktool list
Found number certificates.
1. (X.509 certificate)
Label:  label-name
ID: fingerprint that binds certificate to private key
Subject: subject-DN
Issuer:  distinguished-name
Serial:  hex-serial-number

2. ...
```

Example 26 Importing a PKCS #12 File Into Your Keystore

In the following example, the user imports a PKCS #12 file from a third party. The `pktool import` command extracts the private key and the certificate from the `gracedata.p12` file and stores them in the user's preferred keystore.

```
$ pktool import keystore=pkcs11 infile=gracedata.p12 label=GraceCert
Enter password to use for accessing the PKCS12 file:    Type PKCS #12 password
Enter pin for Sun Software PKCS#11 softtoken:    Type PIN for token
Found 1 certificate(s) and 1 key(s) in gracedata.p12
$ pktool list
No.  Key Type  Key Len.  Key Label
----------------------------------------------------
Asymmetric public keys:
1    RSA                 GraceCert
Certificates:
1    X.509 certificate
Label: GraceCert
ID: 71:8f:11:f5:62:10:35:c2:5d:b4:31:38:96:04:80:25:2e:ad:71:b3
Subject: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
Issuer: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
Serial: 0x00000010
```

Example 27 Importing an X.509 Certificate Into Your Keystore

In the following example, the user imports an X.509 certificate in PEM format into the user's preferred keystore. This public certificate is not protected with a password. The user's public keystore is also not protected by a password.

```
$ pktool import keystore=pkcs11 infile=somecert.pem label="TheirCompany Root Cert"
$ pktool list
No.  Key Type  Key Len.  Key Label
Certificates:
1    X.509 certificate
Label: TheirCompany Root Cert
ID: ec:a2:58:af:83:b9:30:9d:de:b2:06:62:46:a7:34:49:f1:39:00:0e
Subject: C=US, O=TheirCompany, OU=Security, CN=TheirCompany Root CA
Issuer: C=US, O=TheirCompany, OU=Security, CN=TheirCompany Root CA
Serial: 0x00000001
```

▼ How to Export a Certificate and Private Key in PKCS #12 Format

You can create a file in PKCS #12 format to export private keys and their associated X.509 certificate to other systems. Access to the file is protected by a password.

1. **Find the certificate to export.**

```
$ pktool list
Found number certificates.
1. (X.509 certificate)
Label:  label-name
ID: fingerprint that binds certificate to private key
Subject: subject-DN
Issuer:  distinguished-name
Serial:  hex-serial-number

2. ...
```

2. **Export the keys and certificate.**

   Use the keystore and label from the `pktool list` command. Provide a file name for the export file. If the name contains a space, surround the name with double quotes.

```
$ pktool export keystore=keystore outfile=outfile-name label=label-name
```

3. **Protect the export file with a password.**

   At the prompt, type the current password for the keystore. At this point, you create a password for the export file. The receiver must provide this password when importing the file.

```
Enter pin for Sun Software PKCS#11 softtoken:      Type PIN for token
Enter password to use for accessing the PKCS12 file:     Create PKCS #12 password
```

Tip - Send the password separately from the export file. Best practice suggests that you provide the password out of band, such as during a telephone call.

Example 28 Exporting a Certificate and Private Key in PKCS #12 Format

In the following example, a user exports the private keys with their associated X.509 certificate into a standard PKCS #12 file. This file can be imported into other keystores. The PKCS #11 password protects the source keystore. The PKCS #12 password is used to protect private data in the PKCS #12 file. This password is required to import the file.

```
$ pktool list
No.  Key Type  Key Len.  Key Label
----------------------------------------------------
Asymmetric public keys:
1    RSA                 My Cert
Certificates:
1    X.509 certificate
Label: My Cert
ID: d2:7e:20:04:a5:66:e6:31:90:d8:53:28:bc:ef:55:55:dc:a3:69:93
Subject: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
Issuer: C=US, O=My Company, OU=Security Engineering Group, CN=MyCA
Serial: 0x000001

$ pktool export keystore=pkcs11 outfile=mydata.p12 label="My Cert"
Enter pin for Sun Software PKCS#11 softtoken:      Type PIN for token
Enter password to use for accessing the PKCS12 file:     Create PKCS #12 password
```

The user then telephones the recipient and provides the PKCS #12 password.

How to Generate a Passphrase by Using the `pktool setpin` Command

You can generate a passphrase for an object in a keystore, and for the keystore itself. The passphrase is required to access the object or keystore. For an example of generating a passphrase for an object in a keystore, see Example 28, "Exporting a Certificate and Private Key in PKCS #12 Format," on page 65.

1. **Generate a passphrase for access to a keystore.**

```
$ pktool setpin keystore=nss|pkcs11 [dir=directory]
```

The default directory for key storage is `/var/`*username*.

The initial password for a PKCS #11 keystore is `changeme`. The initial password for an NSS keystore is an empty password.

2. **Answer the prompts.**

   When prompted for the current token passphrase, type the token PIN for a PKCS #11 keystore, or press the Return key for an NSS keystore.

   ```
   Enter current token passphrase:        Type PIN or press the Return key
   Create new passphrase:             Type the passphrase that you want to use
   Re-enter new passphrase:                 Retype the passphrase
   Passphrase changed.
   ```

   The keystore is now protected by *passphrase*. If you lose the passphrase, you lose access to the objects in the keystore.

3. **(Optional) Display a list of tokens.**

   ```
   # pktool tokens
   ```

   The output depends on whether the metaslot is enabled. For more information about the metaslot, see "Concepts in the Cryptographic Framework" on page 12.

   - If the metaslot is enabled, the `pktool token` command generates output similar to the following:

     ```
     ID Slot   Name                          Token Name                       Flags
     --        ---------                     ----------                       -----
     0         Sun Metaslot                  Sun Metaslot
     1         Sun Crypto Softtoken          Sun Software PKCS#11 softtoken   LIX
     2         PKCS#11 Interface for TPM     TPM                              LXS
     ```

   - If the metaslot is disabled, the `pktool token` command generates output similar to the following:

     ```
     ID Slot   Name                          Token Name                       Flags
     --        ---------                     ----------                       -----
     1         Sun Crypto Softtoken          Sun Software PKCS#11 softtoken   LIX
     2         PKCS#11 Interface for TPM     TPM                              LXS
     ```

   In the two output versions, flags can be any combination of the following:

   - `L` – login required
   - `I` – initialized
   - `X` – User PIN expired

- `S` – SO PIN expired
- `R` – Write protected

Example 29 Protecting a Keystore With a Passphrase

The following example shows how to set the passphrase for an NSS database. Because no passphrase has been created, the user presses the Return key at the first prompt.

```
$ pktool setpin keystore=nss dir=/var/nss
Enter current token passphrase:     Press the Return key
Create new passphrase: has8n0NdaH
Re-enter new passphrase: has8n0NdaH
Passphrase changed.
```

▼ How to Generate a Key Pair by Using the `pktool genkeypair` Command

Some applications require a public/private key pair. In this procedure, you create these key pairs and store them.

1. **(Optional) If you plan to use a keystore, create the keystore.**
   - **To create and initialize a PKCS #11 keystore, see "How to Generate a Passphrase by Using the `pktool setpin` Command" on page 65.**
   - **To create and initialize an NSS keystore, see Example 29, "Protecting a Keystore With a Passphrase," on page 67.**
2. **Create the key pair.**

   Use one of the following methods.

   - **Create the key pair and store the key pair in a file.**

     File-based keys are created for applications that read keys directly from files on the disk. Typically, applications that directly use OpenSSL cryptographic libraries require that you store the keys and certificates for the application in files.

Note - The `file` keystore does not support elliptic curve (`ec`) keys and certificates.

```
$ pktool genkeypair keystore=file outkey=key-filename \
[format=der|pem] [keytype=rsa|dsa] [keylen=key-size]
```

`keystore=file`

The value `file` specifies the file type of storage location for the key.

`outkey=`*key-filename*

Specifies the name of the file where the key pair is stored.

`format=der|pem`

Specifies the encoding format of the key pair. `der` output is binary, and `pem` output is ASCII.

`keytype=rsa|dsa`

Specifies the type of key pair that can be stored in a `file` keystore. For definitions, see DSA and RSA.

`keylen=`*key-size*

Specifies the length of the key in bits. The number must be divisible by 8. To determine possible key sizes, use the `cryptoadm list -vm` command.

- **Create the key pair and store it in a PKCS #11 keystore.**

You must complete Step 1 before using this method.

The PKCS #11 keystore is used to store objects on a hardware device. The device could be a Sun Crypto Accelerator 6000 card, a trusted platform module (TPM) device, or a smart card that is plugged into the Cryptographic Framework. PKCS #11 can also be used to store objects in the `softtoken`, or software-based token, which stores the objects in a private subdirectory on the disk. For more information, see the `pkcs11_softtoken(5)` man page.

You can retrieve the key pair from the keystore by a label that you specify.

```
$ pktool genkeypair label=key-label \
[token=token[:manuf[:serial]]] \
[keytype=rsa|dsa|ec]  [curve=ECC-Curve-Name]]\
[keylen=key-size] [listcurves]
```

`label=`*key-label*

Specifies a label for the key pair. The key pair can be retrieved from the keystore by its label.

`token=`*token*[:*manuf*[:*serial*]]

Specifies the token name. By default, it is `Sun Software PKCS#11 softtoken`.

`keytype=rsa|dsa|ec [curve=`*ECC-Curve-Name*`]`

Specifies the keypair type. For the elliptic curve type, optionally specifies a curve name. Curve names are listed as output to the `listcurves` option.

`keylen=`*key-size*

Specifies the length of the key in bits. The number must be divisible by 8.

`listcurves`

Lists the elliptic curve names that can be used as values to the `curve=` option for an `ec` key type.

- **Generate the key pair and store it in an NSS keystore.**

The NSS keystore is used by servers that rely on NSS as their primary cryptographic interface.

You must complete Step 1 before using this method.

```
$ pktool keystore=nss genkeypair label=key-nickname \
[token=token[:manuf[:serial]]] \
[dir=directory-path] [prefix=database-prefix] \
[keytype=rsa|dsa|ec] [curve=ECC-Curve-Name]] \
[keylen=key-size] [listcurves]
```

`keystore=nss`

The value `nss` specifies the NSS type of storage location for the key.

`label=`*nickname*

Specifies a label for the key pair. The key pair can be retrieved from the keystore by its label.

`token=`*token*[:*manuf*[:*serial*]]

Specifies the token name. By default, it is `Sun Software PKCS#11 softtoken`.

`dir=`*directory*

Specifies the directory path to the NSS database. By default, *directory* is the current directory.

`prefix=`*database-prefix*

Specifies the prefix to the NSS database. The default is no prefix.

`keytype=rsa|dsa|ec [curve=`*ECC-Curve-Name*`]`

Specifies the keypair type. For the elliptic curve type, optionally specifies a curve name. Curve names are listed as output to the `listcurves` option.

`keylen=`*key-size*

Specifies the length of the key in bits. The number must be divisible by 8.

`listcurves`

Lists the elliptic curve names that can be used as values to the `curve=` option for an `ec` key type.

3. **(Optional) Verify that the key exists.**

   Use one of the following commands, depending on where you stored the key.

   - **Verify the key in the *key-filename* file.**

   ```
   $ pktool list keystore=file objtype=key infile=key-filename
   Found n keys.
   Key #1 - keytype:location (keylen)
   ```

   - **Verify the key in the PKCS #11 keystore.**

   ```
   $ pktool list objtype=key
   Enter PIN for keystore:
   Found n keys.
   Key #1 - keytype:location (keylen)
   ```

   - **Verify the key in the NSS keystore.**

   ```
   $ pktool list keystore=nss dir=directory objtype=key
   ```

Example 30 Creating a Key Pair by Using the `pktool` Command

In the following example, a user creates a PKCS #11 keystore for the first time. After determining the key sizes for RSA key pairs, the user then generates a key pair for an application. Finally, the user verifies that the key pair is in the keystore. The user notes that the second occurrence of the RSA key pair can be stored on hardware. Because the user does not specify a `token` argument, the key pair is stored as a `Sun Software PKCS#11 softtoken`.

```
# pktool setpin
Create new passphrase:
Re-enter new passphrase:      Retype password
Passphrase changed.
```

```
$ cryptoadm list -vm | grep PAIR
...
CKM_DSA_KEY_PAIR_GEN       512  3072 .  .  .  .  .  .  .  .  .  X  .  .  .  .
CKM_RSA_PKCS_KEY_PAIR_GEN  256  8192 .  .  .  .  .  .  .  .  .  X  .  .  .  .
...
CKM_RSA_PKCS_KEY_PAIR_GEN  256  2048 X  .  .  .  .  .  .  .  .  X  .  .  .  .
ecc: CKM_EC_KEY_PAIR_GEN,CKM_ECDH1_DERIVE,CKM_ECDSA,CKM_ECDSA_SHA1
$ pktool genkeypair label=specialappkeypair keytype=rsa keylen=2048
Enter PIN for Sun Software PKCS#11 softtoken  :     Type password

$ pktool list
Enter PIN for Sun Software PKCS#11 softtoken  :     Type password
No.      Key Type      Key Len.      Key Label
----------------------------------------------------
Asymmetric public keys:
1        RSA                         specialappkeypair
```

Example 31 Creating a Key Pair That Uses the Elliptic Curve Algorithm

In the following example, a user adds an elliptic curve (`ec`) key pair to the keystore, specifies a curve name, and verifies that the key pair is in the keystore.

```
$ pktool genkeypair listcurves
secp112r1, secp112r2, secp128r1, secp128r2, secp160k1
.
.
.
c2pnb304w1, c2tnb359v1, c2pnb368w1, c2tnb431r1, prime192v2
prime192v3
$ pktool genkeypair label=eckeypair keytype=ec curves=c2tnb431r1
$ pktool list
Enter PIN for Sun Software PKCS#11 softtoken  :     Type password
No.  Key Type  Key Len.  Key Label
----------------------------------------------------
Asymmetric public keys:
1    ECDSA               eckeypair
```

▼ How to Sign a Certificate Request by Using the `pktool signcsr` Command

This procedure is used to sign a PKCS #10 certificate signing request (CSR). The CSR can be in PEM or DER format. The signing process issues an X.509 v3 certificate. To generate a PKCS #10 CSR, see the `pktool(1)` man page.

Before You Begin This procedure assumes that you are a certificate authority (CA), you have received a CSR, and it is stored in a file. For an example of creating a CSR, see Example 32, "Generating a CSR," on page 73.

1. **Collect the following information for the required arguments to the `pktool signcsr` command:**

`signkey`	If you have stored the signer's key in a PKCS #11 keystore, `signkey` is the label that retrieves this private key. If you have stored the signer's key in an NSS keystore or a file keystore, `signkey` is the file name that holds this private key.
`csr`	Specifies the file name of the CSR.
`serial`	Specifies the serial number of the signed certificate.
`outcert`	Specifies the file name for the signed certificate.
`issuer`	Specifies your CA issuer name in distinguished name (DN) format.

For information about optional arguments to the `signcsr` subcommand, see the `pktool(1)` man page.

2. **Sign the request and issue the certificate.**

For example, the following command signs the certificate with the signer's key from the PKCS #11 repository:

```
# pktool signcsr signkey=CASigningKey \
csr=fromExampleCoCSR \
serial=0x12345678 \
outcert=ExampleCoCert2010 \
issuer="O=Oracle Corporation, \
OU=Oracle Solaris Security Technology, L=Redwood City, ST=CA, C=US, \
CN=rootsign Oracle"
```

The following command signs the certificate with the signer's key from a file:

```
# pktool signcsr signkey=CASigningKey \
csr=fromExampleCoCSR \
serial=0x12345678 \
outcert=ExampleCoCert2010 \
issuer="O=Oracle Corporation, \
OU=Oracle Solaris Security Technology, L=Redwood City, ST=CA, C=US, \
CN=rootsign Oracle"
```

3. **Send the certificate to the requester.**

   You can use email, a web site, or another mechanism to deliver the certificate to the requester.

   For example, you could use email to send the `ExampleCoCert2010` file to the requester.

Example 32 Generating a CSR

This example shows two methods to generate a CSR.

- Use the `pktool` command and store the CSR in the PKCS #11 keystore. You must provide the password to the keystore.

```
$ pktool gencsr keystore=pkcs11 label=example3csr \
   keytype=rsa keylen=2048 hash=sha2 \
   format=pem outcsr=/var/tmp/example3.csr-1 \
   subject="CN=example3.company.au, OU=HR Department, O=Example3, L=Sydney, ST=NSW,
 C=AU"
```

- Use the `openssl` command to generate the CSR.

```
$ openssl req -text -noout -in /var/tmp/example3.csr-1
```

▼ How to Manage Third-Party Plugins in KMF

You identify your plugin by giving it a keystore name. When you add the plugin to KMF, the software identifies it by its keystore name. The plugin can be defined to accept an option. This procedure includes how to remove the plugin from KMF.

1. **Install the plugin.**

```
$ /usr/bin/kmfcfg install keystore=keystore-name \
modulepath=path-to-plugin [option="option-string"]
```

   where:

keystore-name	Specifies a unique name for the keystore that you provide.
path-to-plugin	Specifies the full path to the shared library object for the KMF plugin.
option-string	Specifies an optional argument to the shared library object.

2. **List the plugins.**

```
$ kmfcfg list plugin
keystore-name:path-to-plugin [(built-in)] | [;option=option-string]
```

3. **To remove the plugin, uninstall it and verify its removal.**

```
$ kmfcfg uninstall keystore=keystore-name
$ kmfcfg plugin list
```

Example 33 Calling a KMF Plugin With an Option

In the following example, the administrator stores a KMF plugin in a site-specific directory. The plugin is defined to accept a `debug` option. The administrator adds the plugin and verifies that the plugin is installed.

```
# /usr/bin/kmfcfg install keystore=mykmfplug \
modulepath=/lib/security/site-modules/mykmfplug.so
$ kmfcfg list plugin
KMF plugin information:
-----------------------
pkcs11:kmf_pkcs11.so.1 (built-in)
file:kmf_openssl.so.1 (built-in)
nss:kmf_nss.so.1 (built-in)
mykmfplug:/lib/security/site-modules/mykmfplug.so
# kmfcfg modify plugin keystore=mykmfplug option="debug"
# kmfcfg list plugin
KMF plugin information:
-----------------------
...
mykmfplug:/lib/security/site-modules/mykmfplug.so;option=debug
```

The plugin now runs in debugging mode.

Adding CA Certificates to the Oracle Solaris CA Keystore

Oracle Solaris provides a keystore for Certificate Authority (CA) certificate files. To manage the keystore, you restart the SMF `ca-certificates` service after you add or remove certificates from the keystore.

X.509 certificates contain an RSA public key and the key's signer ("CN" or "Subject"). The key and signer verifies that some file or object was signed with the key holder's private key. CA certificates are issued by well-known organizations to verify that a certificate is legitimate and that the public key in the certificate can be trusted.

Oracle Solaris keeps the CA certificates in the `/etc/certs/CA` directory. Hashed links to the CA certificates are in the `/etc/openssl/certs` directory to enable fast lookup and access, typically by OpenSSL. Usually, each filename in the `/etc/certs/CA` directory is the certificate holder's CN with spaces replaced by underscores ("_") and appended with a `.pem` extension. For example, the file `/etc/certs/CA/ExampleCo-_G3.pem` contains the certificate for CN "ExampleCo Class 4 Public Primary Certification Authority - G3".

If you need to verify a certificate with a CA certificate that is missing, you can add the certificate manually. See "How to Add a CA Certificate to the Oracle Solaris CA Keystore" on page 75.

▼ How to Add a CA Certificate to the Oracle Solaris CA Keystore

Before You Begin You must assume the `root` role. For more information, see "Using Your Assigned Administrative Rights" in *Securing Users and Processes in Oracle Solaris 11.3*.

1. **Verify that the CA certificate is legitimate.**

   Check with the issuer of the CA certificate directly.

   Caution - Do not rely on verification from an entity that did not issue the CA certificate. Do not install invalid CA certificates on your system that your software would treat as trustworthy.

2. **Strip extra text from the certificate.**

   Remove any text that surrounds the "`-----BEGIN CERTIFICATE-----`" and "`-----END CERTIFICATE-----`" lines. Some applications are not able to handle the extra text.

3. **Verify that the certificate is not corrupt.**

   For example, display the text of a certificate by using the `openssl` command.

   ```
   # openssl x509 -noout -text -in Example_Root_CA.pem
   ```

   The output should display the issuer, owner (Subject/DN), validity dates, signature algorithm, and public key, among other information.

4. **Verify that the certificate file is world-readable.**

   If it is not, use the `chmod` command to make the file world-readable.

   ```
   # chmod a+r Example_Root_CA.pem; ls -l Example_Root_CA.pem
   -rw-r--r--   1 root   sys    1500 Sep  10 10:10 Example_Root_CA.pem
   ```

5. **Copy the certificate to the `/etc/certs/CA` directory.**

For example:

```
# cp -p Example_Root_CA.pem /etc/certs/CA/
```

6. **Restart the `ca-certificates` service.**

   ```
   # /usr/sbin/svcadm restart /system/ca-certificates
   ```

   The service adds the certificate to the `/etc/certs/ca-certificates.crt` file and adds a hashed link in the `/etc/openssl/certs` directory.

7. **Verify that the CA certificate service has restarted.**

   When the service restarts, it processes your new CA certificate.

   ```
   $ svcs -x ca-certificates
   svc:/system/ca-certificates:default (CA Certificates Service)
    State: online since Sun Sept 10 10:10:10 2016
      See: openssl(5)
      See: /var/svc/log/system-ca-certificates:default.log
   Impact: None.
   ```

Troubleshooting If the service hasn't started, the certificate could be corrupt or could be a duplicate of an existing CA certificate. Look for error messages in the log file listed in the `svcs -x` command output. Also check the `/system/volatile/system-ca-certificates:default.log` file.

♦ ♦ ♦ CHAPTER 5

KMIP and PKCS #11 Client Applications

Your PKCS #11 applications can now function as clients that use the Key Management Interoperability Protocol (KMIP). These client applications can communicate with KMIP-compliant servers to create and use symmetric keys. Oracle Solaris provides client support for KMIP v1.1: OASIS Standard, enabling clients to communicate with KMIP-compliant servers such as the Oracle Key Vault (OKV).

This chapter covers the following topics:

- "Using KMIP in Oracle Solaris" on page 77
- "KMIP and the Oracle Key Vault" on page 79
- "Benefit of KMIP Support for Oracle Solaris Clients" on page 80

Using KMIP in Oracle Solaris

The new `pkcs11_kmip` provider in the Cryptographic Framework enables PKCS #11 applications to function as KMIP clients and communicate to KMIP-compliant servers. You use the `kmipcfg` command to initialize and manage states of the `pkcs11_kmip` provider.

The `pkcs11_kmip` provider connects PKCS #11 applications to KMIP-compliant servers. In Oracle Solaris, each KMIP *server group* is implemented as a PKCS #11 token plugged into a PKCS #11 slot. The `kmipcfg` command is used to configure the KMIP server groups. The `pktool` command can be used to review the state of these tokens from the PKCS #11 perspective.

To set up KMIP communications for clients in Oracle Solaris, administrators perform the following steps:

1. Install the `pkcs11_kmip` package.

```
$ pkg install pkcs11_kmip
```

This package loads the software provider into the Cryptographic Framework.

2. Create and configure a KMIP server group with the kmipcfg command.

   See configuration examples in the pkcs11_kmip(5) man page and Example 34, "Using kmipcfg to Manage the pkcs11_kmip Provider," on page 78.

What pkcs11_kmip Supports

The pkcs11_kmip provider supports a specific set of PKCS #11 interfaces that are useful during KMIP communications, including interfaces such as C_login, C_OpenSession, and C_CreateObject. To review the full list of supported interfaces, see the pkcs11_kmip(5) man page.

The pkcs11_kmip provider supports symmetric keys with AES algorithms and encryption and decryption operations. The following mechanisms are supported:

- CKM_AES_KEY_GEN
- CKM_AES_CBC_PAD
- CKM_AES_CBC

For further information, see the pkcs11_kmip(5) man page.

Creating and Configuring a KMIP Server Group

The following example shows one option for using the kmipcfg command. For more examples, see the kmipcfg(1M) man page.

EXAMPLE 34 Using kmipcfg to Manage the pkcs11_kmip Provider

This kmipcfg create command creates a server group, cluster1, with three KMIP-compliant servers. The three servers have the following host names:

- server1.example.com
- server2.example.com
- server3.example.com

```
# kmipcfg create \
  -o server_list=server1.example.com,server2.example.com,server3.example.com \
  -o client_p12=cluster1_cred.p12 \
  -o failover_limit=3 cluster1
```

Note the following:

- Each `-o` option specifies one property in the server group configuration. See the `kmipcfg`(1M) man page for a full list of configuration properties.
- Since the port numbers for the servers in this example are not specified, the default port `5696` will be used.
- In this example, the credentials that authenticate and secure the communication are provided in the `cluster1_cred.p12` PKCS #12 bundle. For more information about managing certificates, see the `pktool`(1) man page.
- In this example, if one server in the group fails, the connection will fail over to the next server defined in the `server_list` property. The `failover_limit` property specifies that up to three failovers will be possible.
- This example is non-interactive. For an interactive example, see the `kmipcfg`(1M) man page.

After you create at least one server group, use the `kmipcfg list` command to view configured parameters for the server groups, as in:

```
# kmipcfg list
Server group: cluster1
State: enabled
Hosts:  server1.example.com:5696
        server2.example.com:5696
        server3.example.com:5696
Connection timeout: 5
Cache object time to live: 300
Encoding: TTLV
Failover limit: 3
Client keystore: /var/user/testuser/kmip/cluster1
Client PKCS#12 bundle: cluster1_cred.p12
Secondary authentication type: none
```

KMIP and the Oracle Key Vault

KMIP version 1.1, enables KMIP clients to communicate with KMIP-compliant servers such as the Oracle Key Vault (OKV). To communicate with the OKV, you must first integrate the Oracle Solaris KMIP client with the OKV. In the terminology of the OKV, the Oracle Solaris system must be set up as an OKV *endpoint*.

For instructions, see the following sections in the Oracle Key Vault Administrator's Guide:

- Task 1: Enroll and Provision the Endpoint
- In Special Notes About Endpoint Provisioning, see the sub-section, "If the Endpoint Does Not Use the Oracle Key Vault Client Software".

Benefit of KMIP Support for Oracle Solaris Clients

In Oracle Solaris, KMIP client support provides the following advantages:

- KMIP is an industry protocol. KMIP support enables clients to communicate to any server that is KMIP-compliant. In Oracle Solaris, you can use your PKCS #11 applications as KMIP clients. By connecting these applications to KMIP-compliant servers, you reduce the costs and complexity of key management.

  Note - See "What `pkcs11_kmip` Supports" on page 78 for information about the specific PKCS #11 interfaces and mechanisms that are supported in this release.

- With KMIP server groups, you can ensure that a failed connection to a KMIP server will be passed on and completed by one of the backup servers in that group.
- With multiple server groups, your KMIP clients can open and run multiple KMIP sessions simultaneously. You can access keys from different KMIP-compliant servers on multiple hosts at the same time.

Cryptographic Services Glossary

These glossary entries cover words that can be ambiguous because they are used differently in different parts of the operating system, or have meanings in Oracle Solaris that are distinct from other operating systems.

consumer	In the Cryptographic Framework feature of Oracle Solaris, a consumer is a user of the cryptographic services that come from providers. Consumers can be applications, end users, or kernel operations. Kerberos, IKE, and IPsec are examples of consumers. For examples of providers, see provider.
cryptographic primitive	See primitive.
hardware provider	In the Cryptographic Framework feature of Oracle Solaris, a device driver and its hardware accelerator. Hardware providers offload expensive cryptographic operations from the computer system, thus freeing CPU resources for other uses. See also provider.
MAC	1. A message authentication code (MAC). 2. Also called labeling. In government security terminology, MAC is Mandatory Access Control. Labels such as Top Secret and Confidential are examples of MAC. MAC contrasts with DAC, which is Discretionary Access Control. UNIX permissions are an example of DAC. 3. In hardware, the unique system address on a LAN. If the system is on an Ethernet, the MAC is the Ethernet address.
mechanism	1. A software package that specifies cryptographic techniques to achieve data authentication or confidentiality. Examples: Kerberos V5, Diffie-Hellman public key. 2. In the Cryptographic Framework feature of Oracle Solaris, an implementation of an algorithm for a particular purpose. For example, a DES mechanism that is applied to authentication, such as CKM_DES_MAC, is a separate mechanism from a DES mechanism that is applied to encryption, CKM_DES_CBC_PAD.
password policy	The encryption algorithms that can be used to generate passwords. Can also refer to more general issues around passwords, such as how often the passwords must be changed, how many

	password attempts are permitted, and other security considerations. Security policy requires passwords. Password policy might require passwords to be encrypted with the AES algorithm, and might make further requirements related to password strength.
policy	Generally, a plan or course of action that influences or determines decisions and actions. For computer systems, policy typically means security policy. Your site's security policy is the set of rules that define the sensitivity of the information that is being processed and the measures that are used to protect the information from unauthorized access. For example, security policy might require that systems be audited, that devices must be allocated for use, and that passwords be changed every six weeks. For the implementation of policy in specific areas of the Oracle Solaris OS, see policy in the Cryptographic Framework and password policy.
policy for public key technologies	In the Key Management Framework (KMF), policy is the management of certificate usage. The KMF policy database can put constraints on the use of the keys and certificates that are managed by the KMF library.
policy in the Cryptographic Framework	In the Cryptographic Framework feature of Oracle Solaris, policy is the disabling of existing cryptographic mechanisms. The mechanisms then cannot be used. Policy in the Cryptographic Framework might prevent the use of a particular mechanism, such as `CKM_DES_CBC`, from a provider, such as DES.
primitive	A well-established, low-level algorithm that functions as a basic building block in security systems. Primitives are designed to perform single tasks in a highly reliable fashion.
provider	In the Cryptographic Framework feature of Oracle Solaris, a cryptographic service that is provided to consumers. PKCS #11 libraries, kernel cryptographic modules, and hardware accelerators are examples of providers. Providers plug in to the framework, so are also called *plugins*. For examples of consumers, see consumer.
rights	An alternative to the all-or-nothing superuser model. User rights management and process rights management enable an organization to divide up superuser's privileges and assign them to users or roles. Rights in Oracle Solaris are implemented as kernel privileges, authorizations, and the ability to run a process as a specific UID or GID. Rights can be collected in a rights profile.
rights profile	Also referred to as a profile. A collection of security overrides that can be assigned to a role or user. A rights profile can include authorizations, privileges, commands with security attributes, and other rights profiles that are called supplementary profiles.
security mechanism	See mechanism.
security policy	See policy.

software provider	In the Cryptographic Framework feature of Oracle Solaris, a kernel software module or a PKCS #11 library that provides cryptographic services. See also provider.
superuser model	The typical UNIX model of security on a computer system. In the superuser model, an administrator has all-or-nothing control of the system. Typically, to administer the system, a user becomes superuser (`root`) and can do all administrative activities.
swrand	Entropy provider in kernel. Both kernel and userland have a NIST approved DRBG (Deterministic Random Bit Generator). See NIST Special Publication 800-90A.

Index