

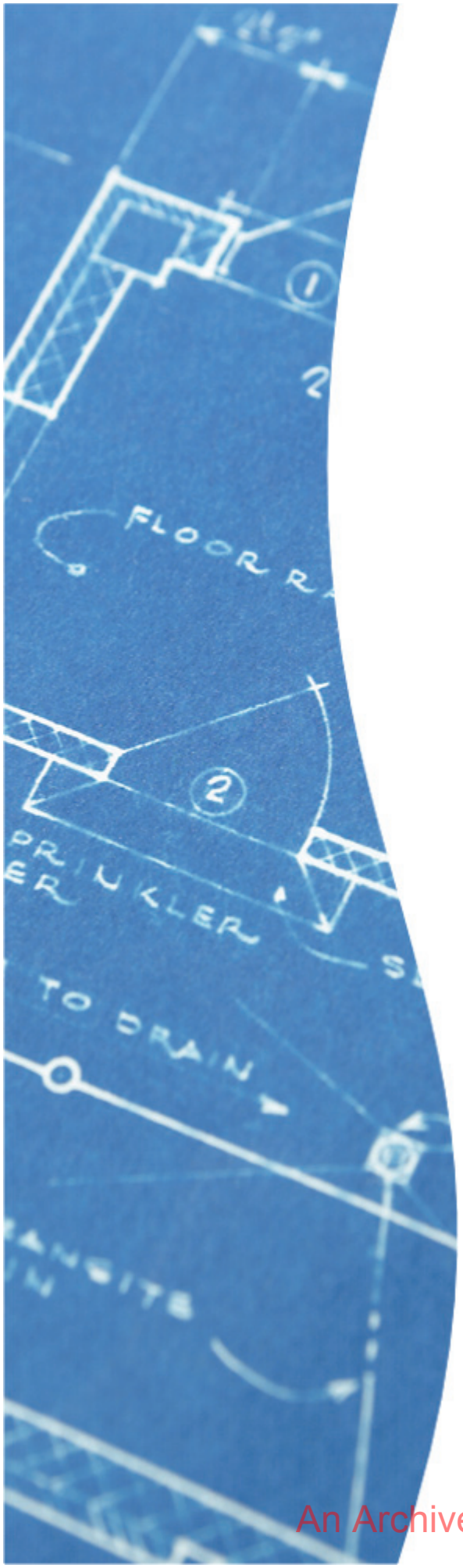
An Archived Oracle Technical Paper
November 2007

USING THE CRYPTOGRAPHIC ACCELERATORS IN THE ULTRASPARC T1 AND T2 PROCESSORS

Important note: this paper was originally published before the acquisition of Sun Microsystems by Oracle in 2010. The original paper is enclosed and distributed as-is. It refers to products that are no longer sold and references technologies that have since been re-named.

ORACLE

An Archived Oracle Technical Paper



USING THE CRYPTOGRAPHIC ACCELERATORS IN THE ULTRASPARC® T1 AND T2 PROCESSORS

Ning Sun
Chi-Chang Lin
Performance and Application Engineering

Sun BluePrints™ Online—November 2007

Part No 819-5782-20
Revision 2.0, 11/27/07
Edition: November 2007

An Archived Oracle Technical Paper

Table of Contents

Cryptography and the Secure Sockets Layer . . . 2
UltraSPARC T1 Processor . . . 2
UltraSPARC T2 Processor . . . 3
Common Performance Capabilities . . . 5
Introduction to the Secure Sockets Layer Protocol . . . 6
Solaris Cryptographic Framework Overview . . . 7
Terminology . . . 7
Solaris Cryptographic Framework Components . . . 8
Administration Utilities . . . 9
Provider Selection Policy . . . 10
UltraSPARC T1 and T2 Processor Cryptographic Provider . . . 10
General Metaslot Configuration . . . 11
General NCP and N2CP Configuration . . . 12
Confirming NCP and N2CP Operation . . . 14
Using NCP and N2CP with Apache . . . 14
Apache with PKCS#11-Based OpenSSL . . . 14
Key and Certificate Installation . . . 16
Tips . . . 17
Using NCP and N2CP with the Sun Java™ System Web Server Software . . . 18
Using pktool to Generate Self-Signed Certificates for Sun Java System Web Server Software . . . 23
Using NCP or N2CP with Java™ Technology Applications . . . 24
Kernel SSL Proxy and NCP . . . 26
How the Solaris Kernel SSL Proxy Works . . . 27
The UltraSPARC T1 and T2 Processors and KSSL Advantage . . . 29
Configuration . . . 30
Performance . . . 32
HTTPS Throughput on UltraSPARC T1 and T2 Processor-Based Servers . . . 32
SPECweb®2005 World Record . . . 33
About the Authors . . . 35
Acknowledgments . . . 35
References . . . 35
Related References . . . 35
Ordering Sun Documents . . . 36
Accessing Sun Documentation Online . . . 36

Introduction to Cryptography and the Secure Sockets Layer . . . 37
Cryptography . . . 37
Secure Sockets Layer . . . 38

An Archived Oracle Technical Paper

Using the Cryptographic Accelerators in the UltraSPARC® T1 and T2 Processors

Cryptographic operations are compute-intensive. Applications that perform frequent cryptographic operations, such as Web servers, typically are deployed in systems incorporating cryptographic accelerator cards, or behind a Secure Sockets Layer (SSL) proxy appliance to offload cryptographic operations and save system CPU cycles for application business logic. These strategies result in complex deployment scenarios.

To address these concerns, Sun released systems based on the low power, eight core, UltraSPARC® T1 processor — a processor targeted at throughput applications that is equipped with built-in hardware cryptographic units to radically simplify and accelerate cryptographic operations. Today, many cryptographic-intensive applications can take advantage of the cryptographic features of Sun Fire T1000, Sun Fire T2000, Sun SPARC Enterprise T1000, and Sun SPARC Enterprise T2000 servers with UltraSPARC T1 processors and reduce costs while reaping performance benefits.

With the advent of a new wave of Internet technology — Web 2.0 — businesses in every industry are even more concerned about secure communications and data privacy. To meet the ever increasing demand on cryptographic operations, Sun released the powerful UltraSPARC T2 processor. With a true System-on-a-Chip (SOC) design, the UltraSPARC T2 processor incorporates additional cryptographic features as well as on-chip I/O and networking capabilities. The latest Sun servers feature this innovative processor, including Sun Fire T5120 and T5220 servers, and Sun Blade™ T6320 server modules.

The cryptographic capabilities of the UltraSPARC T1 and T2 processors can be accessed via the Solaris™ Cryptographic Framework (SCF). SCF provides cryptographic services for kernel-level and user-level consumers, as well as several software encryption modules. SCF continues to include Kernel SSL proxy (KSSL), which offloads SSL processing from user applications and enables them to transparently take advantage of powerful hardware accelerators, like those available in the UltraSPARC T1 and T2 processors.

This Sun BluePrints™ article demonstrates how the combination of the Solaris 10 Operating System (Solaris 10 OS) and the UltraSPARC T1 and T2 processors can be used in a secure Web site. It provides a brief overview of SSL technology, as well as an introduction to the Solaris Cryptographic Framework. The remainder of the document describes how these cryptographic features can be configured and used by common security applications, such as Apache, the Sun Java™ System Web Server, and secure Java technology applications. An earlier version of this Sun BluePrints article detailed the cryptographic capabilities of the UltraSPARC T1 processor. This updated version includes discussions of the new cryptographic features of the UltraSPARC T2 processor.

An Archived Oracle Technical Paper

Cryptography and the Secure Sockets Layer

Cryptography is the study of mathematical techniques focused on information security, including confidentiality, data integrity, and entity and data origin authentication. A set of compute-intensive algorithms typically comprise a cryptography implementation, and can be used by applications when encrypting, decrypting, and hashing data. Some implementations include authentication and verification techniques.

One well-known application of cryptography technology is the implementation of the Secure Sockets Layer (SSL) protocol. Originally developed by Netscape Communications, SSL is a set of rules governing authentication and encrypted communication between servers and clients. As the growth of secure Web site deployment rose rapidly in recent years, the SSL protocol emerged as the *de facto* standard for secure electronic commerce (e-commerce). Indeed, the SSL protocol is now built into all popular Web browsers. Due to the compute-intensive nature of SSL technology, it is anticipated that many of these sites struggle as demand rises, as large volumes of SSL traffic can impact the performance of even the most powerful, general purpose Web server systems.

UltraSPARC T1 Processor

Traditional system designs focus on speeding a single thread of execution, with most processors providing a combination of two threads per core, and at most four cores per chip. Sun's Chip Multithreading initiative represents a paradigm shift that aims to maximize the overall throughput of key commercial workloads. Chip multithreading (CMT) processor technology is key to this approach. The UltraSPARC T1 processor combines chip multiprocessing (CMP) and hardware multithreading (MT) with an efficient instruction pipeline to enable chip multithreading. The resulting processor design provides multiple physical instruction execution pipelines and several active thread contexts per pipeline. Indeed, the UltraSPARC T1 processor raises the limits set by traditional designs, providing four threads per core and up to eight cores per chip with the ability to execute eight instructions in parallel. With up to 32 threads (strands) per chip, the UltraSPARC T1 processor takes thread-level parallelism to a new level.

Figure 1 illustrates the design of the UltraSPARC T1 processor. In this design, each core contains an 8 KB data cache and a 16 KB instruction cache. A four bank, 3 MB unified L2 cache is shared by the eight cores. Each core is capable of running at speeds up to 1.4 GHz, and contains an instruction execution pipeline. A built-in mechanism switches between the four threads of a core such that a new thread is scheduled on the pipeline at each clock cycle in a round robin manner. Four DDR2 channels provide a maximum memory configuration of 64 GB. A single Floating Point Unit (FPU) is shared by the eight cores. The eight cores, L2-cache, FPU, and memory controllers are connected via an on-chip crossbar interconnect. To help speed the Rivest Shamir Adleman (RSA) and Digital Signature Algorithm (DSA) operations needed for SSL processing, each core contains a modular arithmetic unit (MAU) that supports modular exponentiation and multiplication.

An Archived Oracle Technical Paper

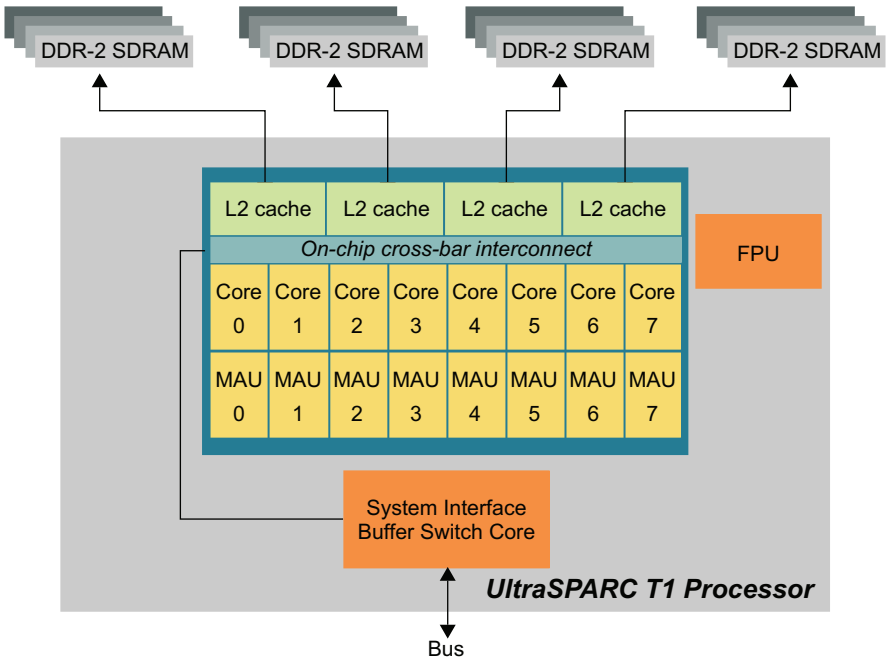


Figure 1. UltraSPARC T1 processor design

UltraSPARC T2 Processor

Keeping the momentum of Sun's Throughput Computing initiative, the UltraSPARC T2 processor is Sun's next generation processor offering. This new processor includes all the features of UltraSPARC T1 processor with double the number of strands, additional cryptographic features and on-chip I/O and on-chip 10 Gigabit Ethernet networking, making the dream of a system-on-chip a reality. Just like the UltraSPARC T1 processor, the UltraSPARC T2 processor combines chip multiprocessing (CMP) and hardware multithreading (MT) with the same eight cores per chip, two instruction execution pipelines per core, and up to four thread contexts per pipeline. With up to eight cores per chip and eight threads per core, totalling to 64 threads per chip, the UltraSPARC T2 processor boasts over 2x performance when compared to the UltraSPARC T1 processor.

Figure 2 illustrates the design of the UltraSPARC T2 processor. Each core can run at speeds up to 1.4 Ghz, and contains an 8 KB data cache and a 16 KB instruction cache. An eight bank, 4 MB unified L2 cache is shared by the eight cores. Each core contains dual pipelines, and a mechanism to switch between the four threads on each pipeline of a core such that a new thread is scheduled on the pipeline at each clock cycle in a round robin manner. Four dual channel Fully Buffered DIMM (FBDIMM) controllers provide a maximum memory configuration of 64 GB, delivering over 60 GB of memory bandwidth. The processor also includes eight Floating Point Units (FPU), with a fully pipelined FPU per core. The eight cores, L2 cache, and memory controllers are connected via an on-chip crossbar interconnect.

An Archived Oracle Technical Paper

To help speed the RSA and DSA operations needed for SSL processing, each core contains a modular arithmetic unit (MAU) that supports modular exponentiation and multiplication. In addition each core contains a Streams Processing Unit (SPU) to speed cryptographic operations. The combination of the MAU and SPU per core makes the UltraSPARC T2 processor an excellent cryptographic engine capable of handling Data Encryption Standard (DES), Triple DES (3DES), Advanced Encryption Standard (AES), Rivest Cipher 4 (RC4), Secure Hash Algorithm SHA1, SHA256, Message Digest algorithm MD5, Rivest Shamir Adleman (RSA) to 2048 key, and Elliptic Curve Cryptosystem (ECC) encryption algorithms.

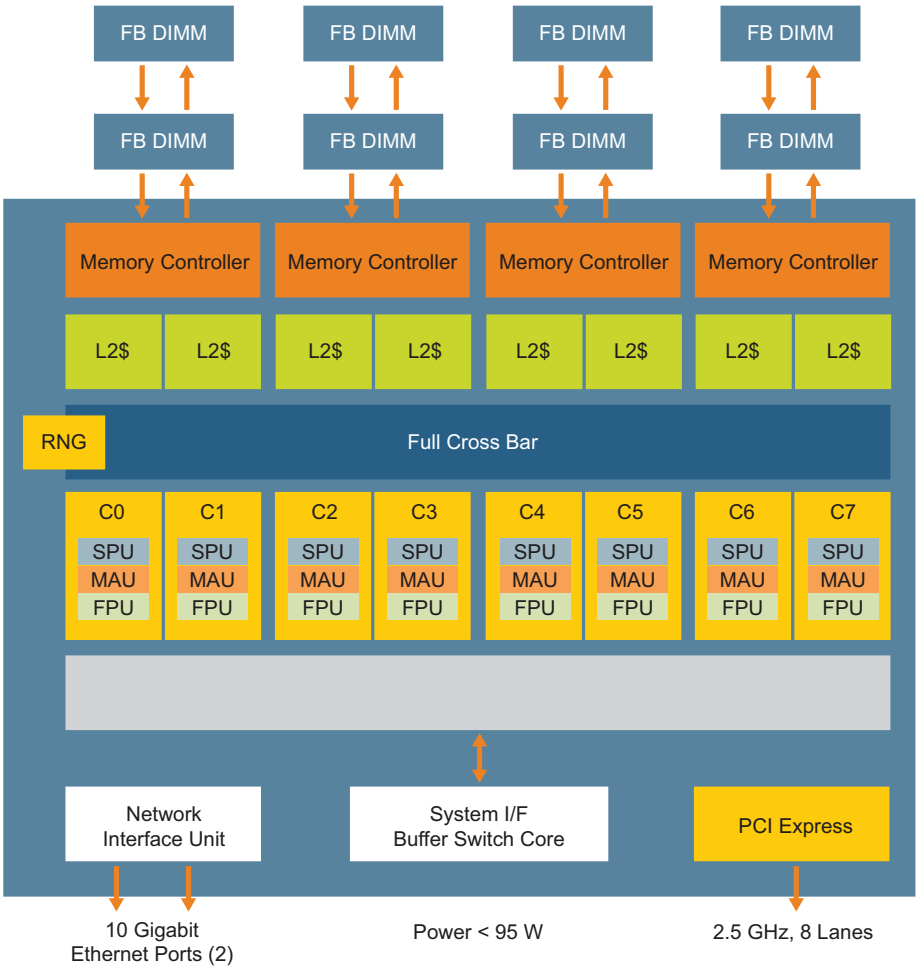


Figure 2. UltraSPARC T2 processor design

An Archived Oracle Technical Paper

Common Performance Capabilities

Traditional data center applications, like Web servers and on-line transaction processing systems, move vast quantities of data in to, and out of, a server. As a result, simply running these applications on systems with fast CPUs and massive instruction level parallelism is not enough to help application performance. Today, memory speed continues to lag behind CPU speed. Consequently, traditional system designs often waste approximately 75 percent of a CPU cycle waiting for a memory transaction to complete. During this time, the execution unit can be kept busy if another thread is ready to execute. This idea is at the heart of the UltraSPARC T1 and T2 processor designs. With four or eight threads per core, the core can be kept busy approximately 85 percent of the time when running a traditional enterprise workload.

The UltraSPARC T2 processor includes an on-chip PCI Express x8 root complex and two 10 Gigabit per second network engines with packet classification and filtering capabilities. The on-chip I/O and network capabilities significantly reduce data movement overhead compared to traditional systems, and is a very good fit for high bandwidth secure network applications like IPSec. To further assist in entropy data generation, the UltraSPARC T2 processor is also armed with an on-chip Random Number Generator (RNG) shared by the eight cores.

Note that the UltraSPARC T1 and T2 processors are SPARC® Version 9 compliant, and can run SPARC Version 7, 8, and 9 binaries without source code modification or re-compilation.

- The eight MAUs, one for each core, are driven by the Niagara Crypto Provider (NCP) device driver in the Solaris 10 OS for both UltraSPARC T1 and T2 processors. On systems with UltraSPARC T1 processors, NCP supports hardware assisted acceleration of RSA and DSA cryptographic operations. On systems with UltraSPARC T2 processors, NCP supports RSA, DSA, DH, and ECC cryptographic operations. NCP is responsible for distributing the load between the MAUs to reduce programming complexity. NCP is pre-configured and enabled in the Solaris Cryptographic Framework on systems with UltraSPARC T1 or T2 processors.
- The eight SPUs (one per core) on the UltraSPARC T2 processor are driven by the Niagara2 Crypto Provider (N2CP) device driver. N2CP supports hardware assisted acceleration for DES, 3DES, AES, RC4, SHA1, SHA256, MD5, ECC and CRC32 algorithms. N2CP is responsible for distributing the load between the SPUs to reduce programming complexity. N2CP is pre-configured and enabled in the Solaris Cryptographic Framework on Sun systems with UltraSPARC T2 processors.
- The Random Number Generator unit is driven by the Niagara2 Random Number Generator (N2RNG) in the Solaris 10 OS.

An Archived Oracle Technical Paper

Introduction to the Secure Sockets Layer Protocol

The Secure Sockets Layer (SSL) protocol relies on a RSA public key cryptographic system for authenticating and securing key exchanges. Each key consists of two parts, called the *public* and *private* keys. Web sites make public keys available to clients, while private keys are known only to servers. During SSL transactions, clients encrypt messages using the server's public key. The server decrypts the message using the private key associated with that public key. Typically, a private key is 1,024 bits long.

The SSL protocol consists of two phases. In the first phase, a client and server agree on a common set of symmetric encryption algorithms and keys to use for security and authentication, called the SSL handshake. In the second phase, these agreed upon mechanisms are used during the actual communication of data messages. Typically, clients, such as Web browsers, initiate the SSL handshake (first phase) after a TCP connection is established. This phase involves a CPU-intensive mathematical operation by the server to recover a secret message shared with the client. Both the client and server perform further operations to derive the encryption, decryption, and message authentication keys from the shared secret.

When a large number of secure clients connect to a secure server, the decryption of the shared secret during the handshake phase consumes a significant portion of available CPU cycles. If the decryption of a shared secret can be offloaded to a specialized device, such as a hardware cryptographic accelerator (known as SSL accelerator cards or SSL co-processors), server CPU resources can focus on handling the business logic of running applications. However, the addition of a specialized cryptographic accelerator card increases system cost and power requirements, and introduces another layer of latency that can significantly impact performance, particularly for small packets. Another technique, used by SSL proxy appliances and SSL proxy switches, terminates SSL connections outside the server. This technique enables these appliances to assume the responsibility of cryptographic operations, enabling the server to spend valuable CPUs cycles on application business logic. Such mechanisms tend to add cost and complexity to the datacenter, and can expose unencrypted data within the network.

During normal SSL operations, application data is exchanged between a client and server in encrypted format, along with a hash called the Message Authentication Code (MAC). The encryption and hash algorithm are agreed on during the handshake phase. The server decrypts the data sent by the client, performs a hash operation on the data, and compares the hash (MAC) value with the one sent by the client. The MAC is computed as `HASH[SHARED-SECRET, ACTUAL-DATA, PADDING-DATA, SEQUENCE-NUMBER]`. When the server sends data back to the client, it encrypts the application data and adds the MAC for the client to validate. Depending on the encryption and hash algorithms agreed upon by the server and client, the encryption, decryption, and hashing process can be CPU-intensive. If these tasks can be offloaded to a hardware cryptographic accelerator, CPU cycles can be saved, improving application scalability.

An Archived Oracle Technical Paper

Sun systems with UltraSPARC T1 or T2 processors do not require the use of additional SSL cards or co-processors. Furthermore, the Solaris 10 OS provides in-kernel SSL termination, a technique that offers greater security than SSL termination outside the server. The remainder of this article describes the security features available in the Solaris 10 OS and how they can be used to build highly scalable, secure, and robust Web sites.

Solaris Cryptographic Framework Overview

SCF provides a set of cryptographic services for kernel-level and user-level consumers. For applications to offload cryptographic operations, the framework also includes several software encryption modules, and gives users the ability to choose either the software encryption module or hardware encryption module for a set of cryptographic operations.

A consistent framework for application-level and kernel-level cryptographic operations, SCF can help increase security and performance while giving applications access to the same hardware accelerators used by the operating system kernel. Based on the PKCS#11 public key cryptography standard created by RSA Security, Inc., SCF provides a mechanism and API whereby both kernel- and user-based cryptographic functions can be executed by the same optimized encryption software or transparently use hardware accelerators configured on the system. This framework brings the power of advanced, streamlined encryption algorithms and hardware acceleration to user-level C and Java programming language-based applications.

Terminology

Cryptographic services utilize the following terms:

- *Consumer*, an application, library, or kernel module that uses or calls cryptographic services.
- *Provider*, an application, library, or kernel module that provides cryptographic services to consumers through the framework.
- *Mechanism*, an entity that implements a cryptographic operation based on an algorithm. Multiple mechanisms may be based on the same algorithm, such that the same algorithm can be used for authentication in one mechanism and for encryption in another.
- *Device*, a hardware or software functional unit that can perform cryptographic operations.
- *Token*, a collection of mechanisms in a record format. A token represents the device in abstract form. Tokens that represent pure software are referred to as *soft tokens*.

An Archived Oracle Technical Paper

- *Slot*, the connecting point for applications to use cryptographic services. A token is plugged into a slot.
- *Key*, a parameter used by an algorithm implementation to produce a specific ciphertext from plaintext using encryption. The number of bits in a key determine the strength of the encryption.
- *Symmetric key cryptography*, a category of encryption in which the same key is used for encryption and decryption. DES, RC4, and AES are examples of symmetric key cryptography.
- *Asymmetric key cryptography*, a type of encryption that uses a pair of keys (a public key, and a private key) for encryption and decryption. One key is used for encryption, and the other for decryption. RSA and DSA are examples of asymmetric key cryptography.
- *Certificate*, a collection of identifying information bound together with a public key and signed by a trusted third-party to prove its authenticity. Certificates and private key objects are stored in the token.
- *Keystore*, a special type of cryptographic device capable of persistent storage for token objects.

See Appendix A for more information about cryptography and the SSL protocol.

Solaris Cryptographic Framework Components

Figure 3 describes the Solaris Cryptographic Framework. Consisting of a user-level framework, and a kernel-level framework, the SCF includes the following components:

- *libpkcs11.so*, the interface user applications can link to in order to call functions in the user-level framework.
- *Pluggable interface*, a PKCS#11-based interface that enables user-level providers to be plugged into the user-level framework.
- *pkcs11_softtoken.so*, user-level cryptographic mechanisms provided by the Solaris OS.
- *pkcs11_softtoken_extra.so*, a user-level library that is identical to *pkcs11_softtoken.so*, but supports stronger keys with bigger key sizes.
- *$HOME/.sunw/pkcs11_softtoken*, the default file system location of the per-user keystore in the Solaris OS.
- *pkcs11_kernel.so*, provides the hardware accelerated algorithms in the kernel-level framework to the user-level framework. Note that it does not contain any cryptography, and does not expose the software-only kernel implementation to the user-level framework.
- *Scheduler and load balancer*, the kernel software responsible for coordinating use, load balancing, and dispatching of the kernel cryptographic service requests.

An Archived Oracle Technical Paper

- *Kernel provider interface*, the interface for kernel-level consumers of cryptographic services. IPSec and Kernel SSL (KSSL) are example consumers.
- *Service provider interface (SPI)*, an interface that enables kernel providers to be plugged into the kernel-level framework. This includes hardware- or software-based cryptographic services.

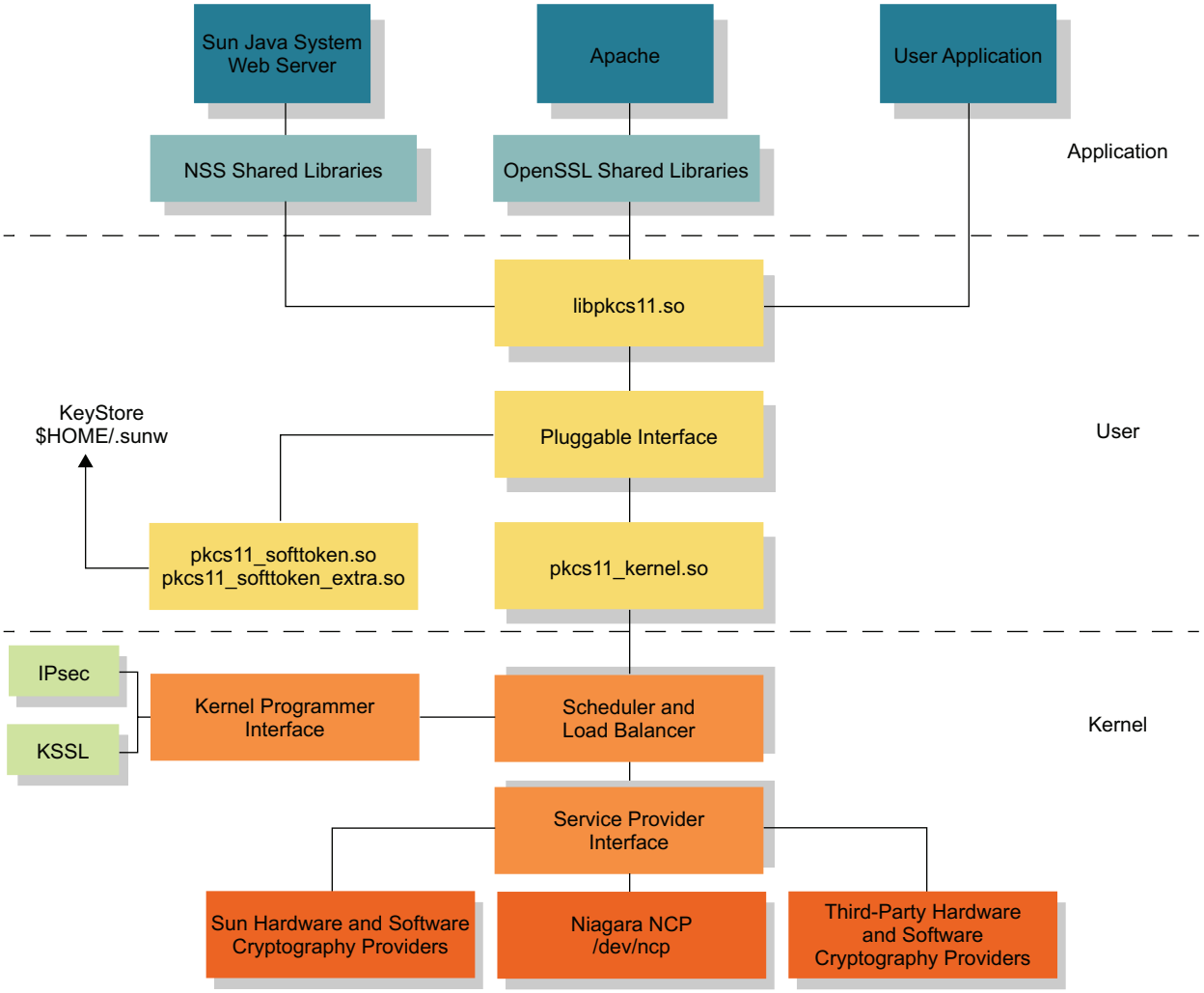


Figure 3. An overview of the Solaris Cryptographic Framework

Administration Utilities

The Solaris Cryptographic Framework provides two administration utilities:

- `cryptoadm(1M)`, a tool that provides administration for both the user-level and kernel-level frameworks. This tool can be used to install hardware and software providers, enable and disable mechanisms for providers, and display cryptographic provider information.

An Archived Oracle Technical Paper

- `pktool(1)`, a tool that lets users manage the certificates and keys on multiple keystores, including PKCS#11 tokens (SCF tokens), Netscape Security Services (NSS) tokens, and standard file-based keystore for OpenSSL. It also provides support to list, delete, and import a Certificate Revocation List (CRL). It does not provide support for creating, signing, or exporting CRLs. The CRL support for the PKCS#11 keystore is file-based.
- The default password for the soft token object store is *changeme*.

Provider Selection Policy

A cryptographic provider can be plugged into the user-level or kernel-level framework if it implements the pluggable cryptographic provider interface based on PKCS#11. Since more than one provider is generally available, the user-level framework exposes a *metaslot* to applications — an abstract slot which acts as the single connection point for the mechanisms provided by all cryptographic providers inserted into the user-level framework. A provider can be inserted into the metaslot with the following command:

```
# cryptoadm install provider=providername
```

A mechanism can have multiple providers. In this case, the order in which providers are inserted into the metaslot determines how lists are searched when a particular mechanism is requested. For example, if *pkcs11_kernel.so* appears before *pkcs11_softtoken.so* in the list, a provider from the kernel-level framework (if available) is chosen over the softtoken provider (*pkcs11_softtoken*) in the user-level framework. This is the default setting in the Solaris OS. To ensure a particular mechanism from a provider is chosen, disable the mechanism from all preceding providers in the metaslot list. The following command can be used to set the metaslot preference for token adoption per mechanism.

```
# cryptoadm enable metaslot [mechanisms=mechanism-list] [token=token-label]
```

UltraSPARC T1 and T2 Processor Cryptographic Provider

RSA operation is an important component of the SSL full handshake. Each core of the UltraSparc T1 and T2 processors includes a Modular Arithmetic Unit (MAU) which supports RSA and DSA operations. RSA operations utilize a compute-intensive algorithm that can be offloaded to the MAU. Indeed, the MAU is capable of sustaining 14,000 RSA operations per second on a system with an UltraSPARC T1 processor, and more than 30,000 RSA operations per second on systems with an UltraSPARC T2 processor. Moving RSA operations to the MAU speeds SSL full handshake performance and frees the CPU to handle business computation. For example, Sun observed a 20 percent to 40 percent improvement in overall throughput in an environment performing approximately 9 percent full handshakes when RSA operations are offloaded to the MAU.

An Archived Oracle Technical Paper

Another aspect of cryptographic operation is the encryption and decryption of sensitive data. Each core of the UltraSPARC T2 processor includes a Streams Processing Unit (SPU) which can perform DES, 3DES, AES, RC4, SHA1, SHA256, MD5, ECC and CRC32 operations. In addition to offloading RSA and DSA operations, compute-intensive encryption and decryption operations, such as 3DES and AES, can also be offloaded on a system incorporating the UltraSPARC T2 processor. Moving these operations to the SPU frees even more CPU cycles. Since most compute-intensive cryptographic operations can now be handled by on-chip hardware, zero cost cryptography is now possible.

The eight MAUs are visible as a single NCP device to consumers, and the eight SPUs are visible as a single N2CP device. Load balancing is performed by the NCP and N2CP drivers. Because the device implementation continues to process requests as long as one MAU or SPU remains functional, systems incorporating UltraSPARC T1 or T2 processors can be considered highly available for cryptographic applications.

The following sections describe how to use the administrative tools of the Solaris Cryptographic Framework to enable and use NCP and N2CP.

General Metaslot Configuration

The soft token and metaslot are easily configured using the following steps.

1. Initialize the user's default keystore by logging in to the system as the application owner.

```
# pktool setpin
Enter token passphrase: changeme
Create new passphrase:
Re-enter new passphrase:
Passphrase changed.
```

2. The *$HOME/.sunw* directory is created as a result of this process. If the application owner does not have a login (such as `nobody`), set the `SOFTTOKEN_DIR` environment variable to a directory with read and write permission by the application owner before proceeding.

```
SOFTTOKEN_DIR=path; export SOFTTOKEN_DIR
```

An Archived Oracle Technical Paper

General NCP and N2CP Configuration

1. Verify the `ncp`, `n2cp`, and `pkcs11_softtoken.so` are available as cryptographic providers by using the `cryptoadm` command. Be sure to run the command as the `root` user. Because NCP is a kernel provider, `pkcs11_kernel.so` should appear before `pkcs11_softtoken.so` in the output. This ordering ensures the framework checks the availability of kernel providers before requesting the `softtoken` implementations.

```
On an UltraSPARC T1 System
# cryptoadm list -p


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.


Kernel software providers:
==========================
des: all mechanisms are enabled.
aes256: all mechanisms are enabled.
arcfour2048: all mechanisms are enabled.
blowfish448: all mechanisms are enabled.
sha1: all mechanisms are enabled.
md5: all mechanisms are enabled.
rsa: all mechanisms are enabled.
swrand: random is enabled.


Kernel hardware providers:
==========================
ncp/0: all mechanisms are enabled
```

An Archived Oracle Technical Paper

```
On an UltraSPARC T2 System
# cryptoadm list -p

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.

Kernel software providers:
==========================
des: all mechanisms are enabled.
aes256: all mechanisms are enabled.
arcfour2048: all mechanisms are enabled.
blowfish448: all mechanisms are enabled.
sha1: all mechanisms are enabled.
sha2: all mechanisms are enabled.
md5: all mechanisms are enabled.
rsa: all mechanisms are enabled.
swrand: random is enabled.

Kernel hardware providers:
==========================
n2cp/0: all mechanisms are enabled.
ncp/0: all mechanisms are enabled.
n2rng/0: all mechanisms are enabled. random is enabled.
```

2. If stronger encryption (greater than 1,024 bit encryption) is desired, install `pkcs11_softtoken_extra.so` using the `cryptoadm` command. Note the `SUNWcry` package must be installed to make `pkcs11_softtoken_extra.so` available on the system.

```
# cryptoadm uninstall provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
# cryptoadm install provider=/usr/lib/security/\$ISA/pkcs11_softtoken_extra.so
```

An Archived Oracle Technical Paper

3. Because the following mechanisms cannot be offloaded to UltraSPARC T1 processors, make sure the following mechanisms are disabled, or disable them in the metaslot.

```
# cryptoadm disable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so \
    mechanism=CKM_SSL3_PRE_MASTER_KEY_GEN, \
    CKM_SSL3_MASTER_KEY_DERIVE, \
    CKM_SSL3_KEY_AND_MAC_DERIVE, \
    CKM_SSL3_MASTER_KEY_DERIVE_DH, \
    CKM_SSL3_MD5_MAC,CKM_SSL3_SHA1_MAC
```

Confirming NCP and N2CP Operation

The `kstat` command below displays the number of RSA public key decryptions performed using NCP since the last system boot. A positive and growing value indicates NCP is operational.

```
# kstat -n ncp0 -s rsaprivate
```

Similarly, the following `kstat` command displays the number of DES, 3DES, AES, RC4, SHA1, SHA256, or MD5 operations performed on systems with UltraSPARC T2 processors using N2CP since the last boot.

```
# kstat -n n2cp0
```

The next few sections describe how to configure applications to run with NCP and N2CP, including Apache, the Sun Java System Web Server, and Java technology applications. Once the application is operational using the Solaris Cryptographic Framework, a set of `kstat` commands can be used to confirm NCP or N2CP is involved in the operation.

Using NCP and N2CP with Apache

The following sections discuss Apache with OpenSSL, key and certificate installation, and important tips to consider.

Apache with PKCS#11-Based OpenSSL

Standard OpenSSL libraries are not built to support the Solaris Cryptographic Framework. However, the default installation of the Solaris 10 OS provides a PKCS#11-based OpenSSL implementation. The Solaris 10 OS includes the following OpenSSL libraries:

- */usr/sfw/lib/libcrypto.so* and */usr/sfw/lib/libssl.so* (for 32-bit applications)
- */usr/sfw/lib/sparcv9/libcrypto.so* and */usr/sfw/lib/sparcv9/libssl.so* (for 64-bit applications)

An Archived Oracle Technical Paper

The *libcrypto.so* shared library calls the *libpkcs11.so* library to bridge OpenSSL-based cryptographic applications with NCP or N2CP. To take advantage of NCP or N2CP, applications must use, or be linked to use, the set of OpenSSL libraries delivered with the Solaris 10 OS in the */usr/sfw* directory. The standard OpenSSL library available from *http://www.openssl.org* cannot take advantage of NCP or N2CP technology.

The following `openssl` command checks the PKCS#11 capability of installed OpenSSL libraries.

```
# /usr/sfw/bin/openssl engine
(pkcs11) PKCS #11 engine support
```

More detailed information can be found by using the `openssl` command with the `-c` and `-t` options.

```
# /usr/sfw/bin/openssl engine -c -t
(pkcs11) PKCS #11 engine support
[RSA, DSA, DH, RAND, DES-CBC, DES-EDE3-CBC, AES-128-CBC,RC4, MD5, SHA1]
[available]
```

The following command runs an `openssl` speed test for RSA operations performed by the NCP.

```
# /usr/sfw/bin/openssl speed -engine pkcs11 rsa
```

On systems with UltraSPARC T2 processors, `openssl` speed tests can be performed for other operations, such as SHA1 performed by the N2CP. The `sha1` option can be replaced with other algorithms, such as `md5`.

```
# /usr/sfw/bin/openssl speed -engine pkcs11 sha1
```

The example below tests the encryption functionality of the UltraSPARC T2 processor using the `openssl` speed test.

```
# /usr/sfw/bin/openssl speed -engine pkcs11 -evp aes-128-cbc
```

The Apache Web Server software should be compiled and linked with the ability to use these OpenSSL libraries. The following options must be supplied to the `configure` command for older Apache software versions, in addition to standard options, in order to compile and build the Apache Web Server software and enable these capabilities.

An Archived Oracle Technical Paper

```
# CFLAGS='-DSSL_EXPERIMENTAL -DSSL_ENGINE' ./configure -enable-ssl \
--enable-rule=SSL_EXPERIMENTAL --with-ssl=/usr/sfw
```

Newer versions of the Apache software, such as Apache 2.2.x, require fewer flags when building the Apache binaries.

```
# CFLAGS='-DSSL_ENGINE' ./configure -enable-ssl --with-ssl=/usr/sfw
```

If the Apache Web Server software was built to use `mod_ssl.so` but did not use the PKCS#11 engine, the `mod_ssl.so` library must be recompiled to include the `SSLCryptoDevice` option. The following options should be used when recompiling the `mod_ssl.so` library.

```
# CFLAGS='-DSSL_EXPERIMENTAL -DSSL_ENGINE' ./configure \
-with-apxs=/usr/local/apache2/bin/apxs \
--enable-rule=SSL_EXPERIMENTAL -with-ssl=/usr/sfw
```

If the `mod_ssl.so` module is built with a newer version of the Apache software, the following flags should be used to ensure connectivity with the Solaris Cryptographic Framework.

```
# CFLAGS='-DSSL_ENGINE' ./configure \
-with-apxs=/usr/local/apache2/bin/apxs -with-ssl=/usr/sfw
```

More information on installing and configuring the Apache software is available at *http://apache.org*

Key and Certificate Installation

Once the software is built and installed, a private key and certificate must be installed for the Apache Web Server software. The following steps outline the process for installing a key and certificate.

1. Generate an RSA private key in Privacy-Enhanced Mail (PEM) format. The key is placed in the */etc/apache/ssl.key/server.key* file.

```
# /usr/sfw/bin/openssl genrsa -des3 -out /etc/apache/ssl.key/server.key 1024
```

2. Generate the certificate request using the key generated.

```
# /usr/sfw/bin/openssl req -new -key /etc/apache/ssl.key/server.key \
-out /etc/apache/ssl.csr/certreq.csr
```

3. Send the certificate (*certreq.csr*) to a certificate authority, such as Verisign, to get the certificate signed.

An Archived Oracle Technical Paper

4. Save the signed certificate in the */etc/apache/ssl.csr/server.crt* file.
5. Edit the *httpd.conf* file and add (or replace) the following lines.

```
SSLCertificateFile /etc/apache/ssl.csr/server.crt
SSLCertificateKeyFile /etc/apache/ssl.key/server.key
```

6. Edit the *ssl.conf* file and add (or replace) the following lines.

```
SSLCryptoDevice pkcs11
```

7. Start the server using the `apachectl` command

```
# /usr/apache/bin/apachectl startssl
```

8. Send a request from a Web browser and check to see the software is working.

The `pktool(1)` utility bundled with the Solaris 10 Operating System can also be used to generate a certificate request and private key. The example below uses the `pktool(1)` command to generate a self-signed private key and certificate and store them in PEM format.

```
# pktool gencert -i keystore=file format=pem outcert=server.crt \
outkey=server.key serial=0x0102030405060708090a0b0c0d0e0f \
lifetime=365-day keylen=1024
```

Tips

The following should be considered when configuring the Apache Web Server software:

- Use of the Apache Web Server software version 2.2.x Prefork mode is recommended for Sun systems with UltraSPARC T1 or UltraSPARC T2 processors.
- The Apache process should not run as the `root` user as a security precaution. Performance may suffer if the software is run as the `root` user and the `pkcs11` engine is enabled.
- The `LogLevel` should be set to `error`.
- Be sure the length of the key is sufficient to meet security needs. Note that longer keys provide greater security but can impact performance.
- Algorithms or mechanisms that should not be offloaded to the SCF should be disabled from SCF using the `cryptoadm(1M)` command. The example below disables the offloading of RC4 to the SCF.

```
# cryptoadm disable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so \mechanism=CKM_RC4
# cryptoadm disable provider=/usr/lib/security/\$ISA/pkcs11_kernel.so \mechanism=CKM_RC4
```

An Archived Oracle Technical Paper

Using NCP and N2CP with the Sun Java™ System Web Server Software

The cryptographic module in the Sun Java System Web Server software is PKCS#11 compliant. As a result, the software can plug into any PKCS#11 compliant cryptographic provider, such as the Solaris Cryptographic Framework. The following steps describe how to configure the Sun Java System Web Server software to use the NCP and N2CP. Slight differences exist between Sun Java System Web Server software versions 6.1 and 7.0. For example, version 6.1 maintains a database at *webserver_instance/../alias/secmod.db* to hold configuration and registry information about security modules. This database is moved to *webserver_instance/config/secmod.db* for version 7.0 of the software. Most steps listed below are the same for the two versions unless otherwise specified. Outputs for some commands for version 7.0 are listed for a better understanding of these instructions. Keep in mind that output from version 6.1 may look slightly different.

1. Set the environment variables for the Sun Java System Web Server software (Sun Java System Web Server 6.1).

```
# cd webserver_instance
# ./start -shell
# LD_PRELOAD=
# LD_PRELOAD_32=
# LD_PRELOAD_64=
# export LD_PRELOAD LD_PRELOAD_32 LD_PRELOAD_64
# cd webserver_instance/../alias
# PATH=webserver_instance/../bin/https/admin/bin:$PATH;export PATH
```

2. Set the environment variables for the Sun Java System Web Server software (Sun Java System Web Server 7.0).

```
# cd webserver_instance
# bin/startserv -shell
# LD_PRELOAD=
# LD_PRELOAD_32=
# LD_PRELOAD_64=
# export LD_PRELOAD LD_PRELOAD_32 LD_PRELOAD_64
```

An Archived Oracle Technical Paper

3. Display the list of modules registered with the Sun Java System Web Server software.

```
# modutil -dbdir . -nocertdb -list
Using database directory ....
Listing of PKCS #11 Modules
-----------------------------------------------------------
1. NSS Internal PKCS #11 Module
         slots: 2 slots attached
        status: loaded

         slot: NSS Internal Cryptographic Services
        token: NSS Generic Crypto Services

         slot: NSS User Private Key and Certificate Services
        token: NSS Certificate DB

2. Root Certs
        library name: libnssckbi.so
         slots: 1 slot attached
        status: loaded

         slot:
        token: Builtin Object Token
-----------------------------------------------------------
```

4. Register the */usr/lib/libpkcs11.so* PKCS11 library with the Sun Java System Web Server software, and enable the slot named `Sun Metaslot`. If a 64-bit version of the Sun Java System Web Server software is used, replace */usr/lib/libpkcs11.so* with */usr/lib/sparcv9/libpkcs11.so* in the command.

```
# modutil -dbdir . -nocertdb -add "Solaris Cryptographic Framework" \t
  -libfile /usr/lib/libpkcs11.so -mechanisms RSA
```

5. On systems with UltraSPARC T2 processors, additional mechanisms can be off-loaded to N2CP.

```
# modutil -dbdir . -nocertdb -add "Solaris Cryptographic Framework" \
-libfile /usr/lib/libpkcs11.so -mechanisms RSA:RC4:AES:DES
```

An Archived Oracle Technical Paper

6. Enable the slot named `Sun Metaslot`.

```
# modutil -dbdir . -nocertdb -disable "Solaris Cryptographic Framework"
# modutil -dbdir . -nocertdb -enable "Solaris Cryptographic Framework" -slot "Sun Metaslot"
```

7. Verify the slot named `Sun Metaslot` was added correctly (UltraSPARC T1).

```
# modutil -dbdir . -nocertdb -list
Using database directory ....
Listing of PKCS #11 Modules
-----------------------------------------------------------
  1. NSS Internal PKCS #11 Module
         slots: 2 slots attached
        status: loaded
         slot: NSS Internal Cryptographic Services
        token: NSS Generic Crypto Services
         slot: NSS User Private Key and Certificate Services
        token: NSS Certificate DB

  2. Solaris Cryptographic Framework
        library name: /usr/lib/libpkcs11.so
         slots: 2 slots attached
        status: loaded
         slot: Sun Metaslot
        token: Sun Metaslot
         slot: ncp/0 Crypto Accel Asym 1.0
        token: ncp/0 Crypto Accel Asym 1.0

  3. Root Certs
library name: libnssckbi.so
         slots: 1 slot attached
        status: loaded
         slot:
        token: Builtin Object Token
-----------------------------------------------------------
```

An Archived Oracle Technical Paper

8. Verify the slot named `Sun Metaslot` was added correctly (UltraSPARC T2).

```
# modutil -dbdir . -nocertdb -list
Using database directory ....
Listing of PKCS #11 Modules
-----------------------------------------------------------
1. NSS Internal PKCS #11 Module
         slots: 2 slots attached
        status: loaded
       slot: NSS Internal Cryptographic Services
        token: NSS Generic Crypto Services
      slot: NSS User Private Key and Certificate Services
        token: NSS Certificate DB
2. Solaris Cryptographic Framework
        library name: /usr/lib/libpkcs11.so
         slots: 4 slots attached
        status: loaded
         slot: Sun Metaslot
        token: Sun Metaslot
         slot: n2cp/0 Crypto Accel Bulk 1.0
        token: n2cp/0 Crypto Accel Bulk 1.0
         slot: ncp/0 Crypto Accel Asym 1.0
        token: ncp/0 Crypto Accel Asym 1.0
         slot: n2rng/0 SUNW_N2_Random_Number_Generator
        token: n2rng/0 SUNW_N2_RNG
3. Root Certs
        library name: libnssckbi.so
         slots: 1 slot attached
        status: loaded
          slot:
        token: Builtin Object Token
-----------------------------------------------------------
```

9. Generate the key and certificate request and get the certificate signed by a Certificate Authority and installed for the `internal` token. If this token was configured previously, skip this step and proceed to step 10 below. See *http://docs.sun.com/source/819-0130-10/agcert.html* for information on configuring the `internal` token. See the next section for instructions on generating and using self-signed certificates.

An Archived Oracle Technical Paper

10. The Sun Java System Web Server software is bundled with the `pk12util` utility. This utility enables users to export certificates and keys from the internal store and import them into an external PKCS#11-based cryptographic service provider.
11. Export the certificate and key from the `internal` store into a PKCS#12 formatted file.

```
For Sun Java System Web Server 6.1:
# pk12util -o certpk12 -d . -n Server-Cert -P webserver_instance
Enter Password or Pin for "internal": password_used_in_setpin
Enter password for PKCS12 file: new_password_for_cert_file

For Sun Java System Web Server 7.0:
# pk12util -o certpk12 -d . -n Server-Cert
Enter Password or Pin for "internal": password_used_in_setpin
Enter password for PKCS12 file: new_password_for_cert_file
```

12. Import the key and certificate in the `Sun Metaslot`.

```
# pk12util -i certpk12 -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot": password_used_in_setpin
Enter password for PKCS12 file: new_password_for_cert_file
```

13. Verify the certificate and key were successfully imported.

```
# certutil -L -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot":
Sun Metaslot:Server-Cert u,u,u
```

14. Configure the Sun Java System Web Server instance to use the metaslot by prepending the original token name with `"Sun Metaslot:"`. For example, replace `"Server-Cert"` with `"Sun Metaslot:Server-Cert"` in the *server.xml* file for the `"server-cert-nickname"`attribute (or `servecertnickname` for version 6.1).
15. Finally, restart the Sun Java System Web Server software. Be sure to specify the password for `Sun Metaslot`, which should be the password specified in the `setpin` command earlier in the process. If the `SOFTTOKEN_DIR` environment variable was used earlier in this procedure, be sure to set it to the same value before starting the server.

An Archived Oracle Technical Paper

Using `pktool` to Generate Self-Signed Certificates for Sun Java System Web Server Software

The following steps explain how to generate self-signed certificates. Follow steps 1 through 8 of the previous section, and proceed with steps 9 through 16 below to generate and use a self-signed certificate.

9. Generate the key and self-signed for the `internal` module of the server. The following `pktool` command is for version 7.0. For version 6.1, a `prefix` flag must be specified. For example, if the Web server instance name is *https-myinstance* and the hostname is *myhost*, add the flag `prefix=https-myinstance-myhost` to the following command.

```
# pktool gencert -i keystore=nss label=cert-pkcs \
serial=0x0102030405060708090a0b0c0d0e0f lifetime=365-day keylen=1024
```

10. Export the certificate and key from the `internal` store into a PKCS#12 formatted file.

```
# pk12util -o certpk12 -d . -n cert-pkcs
Enter Password for PKCS12 file:
Re-enter password:
pk12util: PKCS12 EXPORT SUCCESSFUL
```

11. Import the key and certificate in the `Sun Metaslot`.

```
# pk12util -i certpk12 -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot":
Enter Password for PKCS12 file:
pk12util: PKCS12 IMPORT SUCCESSFUL
```

12. Alternatively, a self-signed key and certificate can be added for the metaslot directly, eliminating the need for the export and import steps outlined in steps 9 and 10 above. In that case, steps 9 and 10 should be replaced with a single step (see step 13 below). If the server's internal modules already had a certificate before configuring the server to use the Solaris Cryptographic Framework, it can be exported and imported into the `Sun Metaslot` using steps 9 and 10 above.

13. Generate the key and self-signed certificate for the `Sun Metaslot`.

```
# pktool gencert -i keystore=pkcs11 label=cert-pkcs \
serial=0x0102030405060708090a0b0c0d0e0f lifetime=365-day keylen=1024
```

An Archived Oracle Technical Paper

14. Verify the certificate and key were successfully imported.

```
# certutil -L -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot":
Sun Metaslot:cert-pkcs u,u,u
```

15. Configure the Sun Java System Web Server instance to use the metaslot by prepending the original token name with "`Sun Metaslot:`". For example, replace "`Server-Cert`" with "`Sun Metaslot:cert-pkcs`" in the *server.xml* file for the "`server-cert-nickname`"attribute.
16. Finally, restart the Sun Java System Web Server software. Be sure to specify the password for `Sun Metaslot`, which should be the password specified in the `setpin` command earlier in the process. If the `SOFTTOKEN_DIR` environment variable was used earlier in this procedure, be sure to set it to the same value before starting the server.

Using NCP or N2CP with Java™ Technology Applications

The first release of the Security API in the Java 2 Platform Standard Edition 1.1 Developer Kit (JDK 1.1) introduced the Java Cryptography Architecture (JCA), a framework for accessing and developing cryptographic functionality for Java platforms. It includes a provider architecture that supports multiple, interoperable cryptography implementations. As part of the Java Secure Socket Extension (JSSE), the Java Cryptography Extension (JCE) extends the JCA API to include APIs for encryption, key exchange, and Message Authentication Code (MAC). The JDK version 1.4 and later includes a JSSE provider (SunJSSE) that is pre-installed and pre-registered with the JCA.

In the Java™ 2 Platform, Standard Edition (J2SE) version 1.5 and later versions, the SunJSSE provider uses JCE exclusively for all of its cryptographic operations. As a result, the SunJSSE provider can automatically take advantage of JCE features and enhancements, including new support for PKCS#11 via the Sun PKCS#11 provider. This enables the SunJSSE provider in J2SE 1.5 to use hardware cryptographic accelerators for significant performance improvements. Therefore, the use of hardware cryptographic accelerators is automatic if:

- JCE is configured to use the Sun PKCS#11 provider
- The Sun PKCS#11 provider is configured to use the underlying accelerator hardware

Unlike many other providers, the Sun PKCS#11 provider does not implement cryptographic algorithms. It is simply a bridge between the Java JCA and JCE APIs, and the native PKCS#11 cryptographic API, translating calls and conventions between the APIs. This means that Java technology applications that call standard JCA and JCE APIs can take advantage of algorithms provided by underlying PKCS#11 implementations without modification. On Sun systems with UltraSPARC T1 or T2 processors, the default

An Archived Oracle Technical Paper

Java Virtual Machine (JVM) and J2SE 1.5 components are preconfigured to use the SunPKCS#11 Provider. The mapping is set up by the following line in the *java-home/jre/lib/security/java.security* file.

```
# List of providers and their preference orders (see above):
security.provider.1=sun.security.pkcs11.SunPKCS11 ${java.home}/lib/
security/sunpkcs11-solaris.cfg
security.provider.2=sun.security.provider.Sun
security.provider.3=sun.security.rsa.SunRsaSign
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
```

The *java-home/jre/lib/security/sunpkcs11-solaris.cfg* file configures the Sun PKCS#11 provider to utilize the Solaris Cryptographic Framework. Mechanisms to be handled by applications can be disabled from the Sun PKCS#11 provider using this file. Sample file content follows.

```
# Configuration file to allow the SunPKCS11 provider to utilize
# the Solaris Cryptographic Framework, if it is available
name = Solaris
description = SunPKCS11 accessing Solaris Cryptographic Framework
library = /usr/lib/$ISA/libpkcs11.so
handleStartupErrors = ignoreAll
attributes = compatibility
disabledMechanisms = {
  CKM_MD2
  CKM_MD5
  CKM_SHA_1
  CKM_SHA256
  CKM_SHA384
  CKM_SHA512
  CKM_DSA_KEY_PAIR_GEN
# KEY_AND_MAC_DERIVE disabled due to Solaris bug 6306708
  CKM_SSL3_KEY_AND_MAC_DERIVE
  CKM_TLS_KEY_AND_MAC_DERIVE
```

An Archived Oracle Technical Paper

```
# the following mechanisms are disabled due to performance issues
# (Solaris bug 6337157)
  CKM_DSA_SHA1
  CKM_MD5_RSA_PKCS
  CKM_SHA1_RSA_PKCS
  CKM_SHA256_RSA_PKCS
  CKM_SHA384_RSA_PKCS
  CKM_SHA512_RSA_PKCS
}
```

On systems with UltraSPARC T2 processors, the *$java-home/jre/lib/security/sunpkcs11-solaris.cfg* file can be modified to remove `CKM_MD5`, `CKM_SHA_1`, `CKM_SHA256` from the disabled mechanisms list (shown in above `disableMechanisms` line) so that these algorithms can be offloaded to N2CP.

Other complex mechanisms, such as `CKM_DSA_SHA1`, `CKM_MD5_RSA_PKCS`, `CKM_SHA1_RSA_PKCS`, `CKM_SHA256_RSA_PKCS`, `CKM_SHA384_RSA_PKCS` and `CKM_SHA512_RSA_PKCS`, cannot be offloaded at this time, although the UltraSPARC T2 processor supports each SHA, MD5, DSA and RSA algorithm individually. To offload these complex mechanisms, JSSE or Metaslot must break these complex mechanisms into simple ones for the *pkcs11_kernel.so* provider to handle individually. A single provider is not currently available that aggregates NCP and N2CP capabilities.

Kernel SSL Proxy and NCP

The Kernel SSL proxy implements the SSL protocol such that a non-SSL application server is able to handle SSL-based client requests. All SSL processing is performed in the kernel. As a result, server programs need only send and receive data in clear text. Figure 4 provides an overview of the Kernel SSL Proxy.

An Archived Oracle Technical Paper

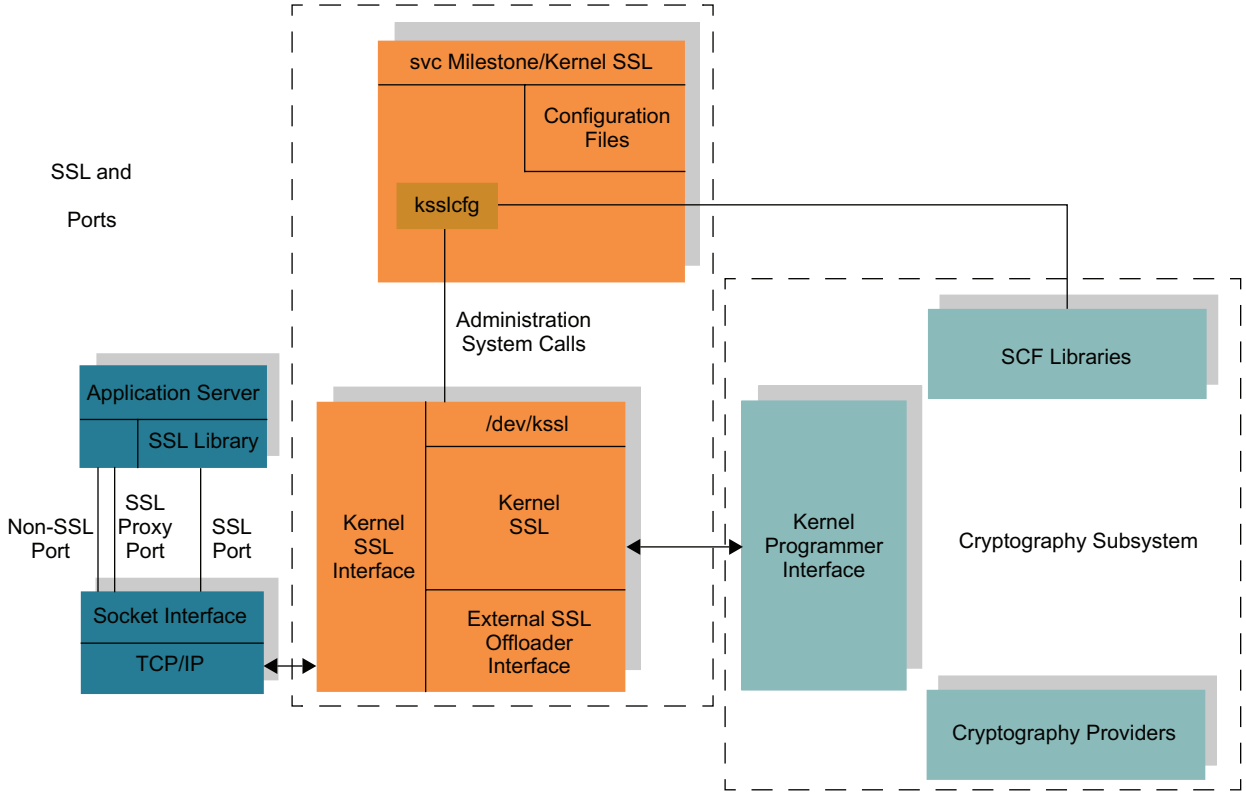


Figure 4. The Kernel SSL Proxy

How the Solaris Kernel SSL Proxy Works

The Solaris Kernel SSL (KSSL) proxy implementation adds a kernel module (`kssl`) which is responsible for the server-side SSL protocol. This module acts as the SSL proxy server, and is responsible for providing a proxy port to the application server and listening on the SSL port. It also manages keys and certificates, manages SSL session state information, and is responsible for the SSL handshake with clients. The SSL handshake, or SSL alert, is handled asynchronously without the application server's knowledge or involvement. However, the encryption, decryption, and message digest is still performed in the context of the application server. The kernel SSL proxy does not implement the full set of cipher suites defined by the Secure Sockets Layer/Transport Layer Security (SSL/TLS) protocol.

For the SSL Proxy module to be active on a given SSL port, such as port 443, the application server must listen on a proxy port, such as port 8080. Three listeners can be configured in the application server:

- One listening on a regular clear text port, such as HTTP port 80.
- One listening on the SSL proxy port through which the `kssl` module sends and receives clear text payloads for secure clients.

An Archived Oracle Technical Paper

- A third port for managing encrypted data at the user level, acting as a user-level SSL server. In other words, the user-level SSL server configuration need not be changed or disabled when configuring KSSL. In such a configuration, client connections requesting a cipher suite that is not supported by KSSL are forwarded to the user-level SSL server as a fallback mechanism. For this mechanism to succeed, the user-level SSL server and KSSL must listen on the same SSL port.

Based on these considerations, three different configurations can be derived:

- One SSL proxy port for clear text communication between the Kernel SSL proxy and the application, and one port for the SSL-protected traffic between the Kernel SSL proxy and remote SSL clients.
- The same configuration as above, with the addition of a fallback listener in the application on the same SSL port for handling the cipher suites not supported by the Kernel SSL proxy. In this case, the application server also behaves as a secure server.
- For application servers that want to offer a mixture of clear text and SSL protected services, another listener can be added on a clear text port that is independent from the proxy port. This is a typical configuration for e-commerce deployment.

When configured, the `kssl` module maintains a table of four-tuples, (IP address, SSL port, proxy port, SSL parameters), where the SSL parameters include the private key, server certificate, acceptable cipher suites, and other SSL session related parameters. The user-level `ksslcfg` command configures the `kssl` module by adding and removing entries from this table.

When a user-level application calls a normal `bind()` operation, the kernel inspects the IP address and port specified in the function call. Several outcomes are possible as a result of this inspection:

- If the IP address and port do not match those configured with the `ksslcfg` command, data for the endpoint is not intercepted or handled by the `kssl` module.
- If the port matches the SSL port configured via the `ksslcfg` command, then the port is used as the fallback port for forwarding all SSL requests that cannot be handled to the user-level SSL server.
- If the port matches the proxy port configured with the `ksslcfg` command, then this binding endpoint acts as the proxy port. The `kssl` module sends and receives clear text data through this endpoint with the application binding to this endpoint. At this time, the `kssl` module starts listening for incoming SSL requests on the SSL port. The `kssl` module guarantees that the application binding to the SSL proxy endpoint will not be attacked at that endpoint with a clear text message, because the proxy port is not accessible from outside the server.

An Archived Oracle Technical Paper

When a client makes a new SSL connection to the endpoint on which the `kssl` module is listening, the client offers a list of cipher suites that it is willing to negotiate. If at least one the cipher suites can be handled by the `kssl` module, the cipher is chosen, the handshake is completed, and the application is notified of an incoming connection at the proxy port. Once the application accepts the new connection, all cryptographic operations on the application payload are handled by the `kssl` module.

When the application performs a `read()` operation, the `kssl` module works in the context of the application to verify the MAC, decrypt the payload, and strip out the SSL header and tail of the incoming record and copy the plain text payload to the user buffer supplied as an argument to the `read()` system call. The process is similar for a `write()` operation. The `kssl` module uses the application context to encrypt and compute the MAC of the outgoing message before actually sending out the encrypted message. In the event no cipher suite matches and a fallback port is not available, the client is notified by the `kssl` module. If the application is also listening on the SSL port endpoint, all connection information is passed to the application as a fallback mechanism for the `kssl` module. At that point, the `kssl` module is no longer interested in the connection.

The UltraSPARC T1 and T2 Processors and KSSL Advantage

Since the `kssl` module is built using the Solaris Encryption Framework kernel APIs, it can take advantage of the NCP module for RSA operation during a handshake and transparently utilize the N2CP module for encryption and hashing operations during a data transfer. KSSL also works with hardware cryptographic cards that can be plugged into the Solaris Cryptographic Framework.

KSSL provides many advantages over other approaches using regular SSL proxy devices that terminate the TCP connection on the device and originate a new TCP connection to the application server. If the SSL proxy device is outside the application server but within the same datacenter, the plain text data moving between the proxy device and the application server hardware can be intercepted, posing a security threat. If the SSL proxy device is an I/O card on the application server, it may not be able to manage a large number of SSL sessions due to limited resource availability on the card, such as memory resources.

The combination of Sun systems with UltraSPARC T1 or T2 processors, NCP or N2CP, and the Solaris OS enable the `kssl` module to take advantage of all the features provided by the operating system and the Solaris Cryptographic Framework. Therefore, KSSL transparently offloads RSA operations to the NCP device on systems with UltraSPARC T1 or T2 processors. On systems incorporating UltraSPARC T2 processors, KSSL offloads RSA operations as well as some commonly used encryption algorithms, such as RC4, DES, and 3DES, along with HMAC (SHA1_HMAC, MD5_HMAC) computation of outbound packets. As is demonstrated later in this article, Sun systems with UltraSPARC T1 or T2

An Archived Oracle Technical Paper

processors configured to use `kssl` offer a robust, scalable, and secure application server solution.

Configuration

The `ksslcfg(1M)` command manages the Service Management Facility (`smf(5)`) instances for the kernel SSL proxy module. It creates and deletes instances of the *svc://network/ssl/proxy* Service Management Facility service. Each instance is unique for a host and SSL port, and specifies the proxy port and SSL parameters to be used by the kernel proxy when the service instance is started. Availability for a given SSL port can be checked with the `svcs` command:

```
# svcs | grep kssl
online         21:08:34 svc:/network/ssl/proxy:kssl-INADDR_ANY-443
```

By default, the `ksslcfg` command creates a Service Management Facility (SMF) instance for SSL port 443, proxy port 8080, and the `INADDR_ANY` IP address. The command assumes the certificate and key file is located in the */etc/kssl/keypair.pem* file (in PEM format), and that passwords for the private key are stored in the */etc/kssl/passphrase* file in clear text.

There are two primary reasons to create a SMF instance:

- Create configuration information for the proxy that is persistent across reboots.
- Provide a service with the Fault Management Resource Identifier (FMRI) that can be started, stopped, and observed by SMF tools, such as the `svcs` and `svcadm` commands, and that can be used by other SMF-enabled services to express dependencies.

If users want to configure the SMF instance manually after each system boot, the service can be disabled using either the `svcadm` or `ksslcfg` command.

```
# svcadm disable svc:/network/ssl/proxy:kssl-INADDR_ANY-443
OR
# ksslcfg delete 443
```

An Archived Oracle Technical Paper

Following is the usage information for the `ksslcfg` command:

```
Usage:

ksslcfg create -f pkcs11 [-d softtoken_directory] -T token_label -C
certificate_subject -x proxy_port [options] [server_address] ssl_port

ksslcfg create -f pkcs12 -i certificate_file -x proxy_port [options]
[server_address] ssl_port

ksslcfg create -f pem -i certificate_file -x proxy_port [options]
[server_address] ssl_port

options are:

        [-c ciphersuites]

        [-p password_file]

        [-t ssl_session_cache_timeout]

        [-u username]

        [-z ssl_session_cache_size]

        [-v]

ksslcfg delete [-v] [server_address] ssl_port
```

The `create` option configures the `kssl` module with a port-pair (SSL port and proxy port). The `delete` option removes a previously configured port-pair. The `ssl_port` is the port to which clients connect, such as HTTPS port 443. The application server must be configured to listen on the `proxy_port`, such as port 8080. If the application server binds to a specific IP address, or a set of IP addresses, then the `ksslcfg` command should be executed with the `server_address` argument, once for each address.

During configuration, the `kssl` module expects a certificate. The certificate can be passed to the `kssl` module during execution of the `ksslcfg` command. The `ksslcfg` command can work with multiple file formats. Note that the `-f` option specifies the file format.

The password for the keypair file is stored in clear text format in a file. This file name is passed as an argument to the `-p` option of the `ksslcfg` command. This option is almost always mandatory when configuring the `kssl` module.

To use the certificate and key from a previously configured softtoken directory, such as *$HOME/.sunw*, use the following ksslcfg command.

```
# ksslcfg create -f pkcs11 [-d softtoken_directory] -T token_label -C  \
   certificate_subject -x proxy_port [options] [server_address] server_port
```

The following steps illustrate how to use a PKCS12 formatted private key and certificate.

An Archived Oracle Technical Paper

1. Create the combined key and certificate file in PKCS12 format. Note that the certificate and key can also be exported from a previously existing store, such as from the store managed by the Sun Java System Web Server software, using the `pk12util` command as described earlier in this article.

```
# /usr/sfw/bin/openssl pkcs12 -export -inkey path_to_private_key_pem_file \
   -in path_to_certificate_pem_file -out keypair_file_in_pk12_format
```

2. Run the `ksslcfg` command.

```
# ksslcfg create  -f pkcs12  -i keypairfile_in_pk12_format -x proxy_port \
                [options] [server_address] server_port
```

The following steps illustrate how to use a key and certificate PEM formatted file.

1. Concatenate the private key and certificate in a single file.

```
# cat private_key.pem certificate.pem > keypair.pem
```

2. Run the `ksslcfg` command.

```
# ksslcfg create -f pem -i keypair.pem -x proxy_port [options] \
   [server_address] server_port
```

Performance

The following sections demonstrate how the Sun systems with UltraSPARC T1 and T2 processors running the Solaris OS work together to deliver superior SSL performance.

HTTPS Throughput on UltraSPARC T1 and T2 Processor-Based Servers

A modified SPECweb®99_SSL workload (*http://www.spec.org/web99ssl/*) was used to perform entirely static HTTPS requests. Designed by the Standard Performance Evaluation Corporation, the SPECweb99_SSL is an industry standard benchmark that measures Web server performance. The benchmark was retired in October, 2005. Because it uses a modified workload, the results presented here cannot be compared with other modified or unmodified SPECweb99_SSL benchmarks performed outside the scope of this document.

The Sun Java System Web Server 6.1 SP5 (64-bit) software and Apache Web Server 2.0.55 (compiled in prefork mode) were used in the study. Table 1 summarizes the results. Note that a Sun Fire T2000 server that uses both KSSL and NCP outperformed the Apache software by 82 percent, and the Sun Java System Web Server 6.1 SP5 (64-bit) doubled its performance than when run without these capabilities. Note that NCP by itself also improves performance.

An Archived Oracle Technical Paper

Table 1. HTTPS throughput results

System	KSSL	NCP	N2CP	Sun Java System Web Server 6.1 SP5 64-bit (Operations/Second)	Apache 2.0.55 Prefork (Operations/Second)
Sun Server With Eight 1.2 Ghz UltraSPARC T1 Processors	Yes	Yes	N/A	13,558	9,406
	No	Yes	N/A	9,548	5,726
	No	No	N/A	6,736	5,288
Sun Server With Eight 1.4 GHz UltraSPARC T2 Processors	Yes	Yes	Yes	29,140	18,911

The last row in the Table 1 shows HTTPS performance on a Sun system with eight UltraSPARC T2 processor cores, each running at 1.4 Ghz. Throughput is more than double that of the systems with UltraSPARC T1 processors. The following factors contribute to this performance boost:

- One extra execution pipeline for each core is available in UltraSPARC T2 processors
- A larger L2 cache is available on UltraSPARC T2 processors
- Improved on-chip RSA performance by the MAU and NCP
- On-chip encryption, decryption, and hashing acceleration performed by N2CP
- On-chip I/O that moves data faster on UltraSPARC T2 processors
- Two on-chip 10 Gbps network connections on UltraSPARC T2 processors

SPECweb®2005 World Record

The latest benchmark available from the Standard Performance Evaluation Corporation (SPEC®), the SPECweb®2005 benchmark measures Web server performance. The benchmark contains multiple standardized workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP). One component of the benchmark focuses on SSL performance. Two sub-workloads use SSL, with the Banking workload consisting entirely of HTTPS requests. More information on the SPECweb2005 benchmark can be found at *htpp://www.spec.org/osg/web2005/*.

Table 2 and Figure 5 compare the Sun SPARC Enterprise T5220 and Sun Fire T2000 servers with other systems. Results are based on *http://www.spec.org/osg/web2005/results/web2005.html* as of October 9, 2007. Only the best numbers for each vendor are described. All results for systems with UltraSPARC T1 and T2 processors were obtained with kernel SSL proxy and NCP or N2CP. The newly released Sun SPARC Enterprise T5220 server incorporating one UltraSPARC T2 processor with eight processor cores sets a new record and outperforms the latest quad-core Intel® systems with 16 cores by more than 20 percent, while significantly reducing both space and power consumption. More detailed analysis can be found at *http://sun.com/servers/coolthreads/t5220/benchmarks.jsp?display=4#4*.

An Archived Oracle Technical Paper

Table 2. SPECweb2005 results for different systems as of October 9, 2007

System	Configuration	Result
Sun SPARC Enterprise T5220 Server	One processor with eight cores running at 1.4 GHz	37,001
Sun SPARC Enterprise T2000 Server	One processor with eight cores running at 1.4 GHz	16,407
Sun Fire T2000 Server	One processor with eight cores running at 1.2 GHz	14,001
Sun Fire T1000 Server	One processor with eight cores running at 1.2 GHz	10,466
Powerleader Science & Technology PR4700D Server	Two processors with eight cores running at 3.0 GHz (Intel)	22,332
HP ProLiant DL580 G5 Server	Four processors with eight cores running at 2.4 GHz (Intel)	26,119
HP ProLiant DL580 G4 Server	Four processors with 16 cores running at 2.93 GHz (Intel)	30,261
HP ProLiant DL580 G2 Server	Four processors with eight cores running at 3.9 GHz (AMD)	22,254
Dell PowerEdge 2950 Server	Two processors with eight cores running at 2.66 GHz (Intel)	16,830
Dell PowerEdge 2950 Server	Two processors with four cores running at 3.0 GHz (Intel)	14,495
Fujitsu Primergy TX300 S3 Server	Two processors with eight cores running at 2.66 GHz (AMD)	18,160
Fujitsu Primergy RX600 S3 Server	Two processors with eight cores running at 3.4 GHz (Intel)	14,896

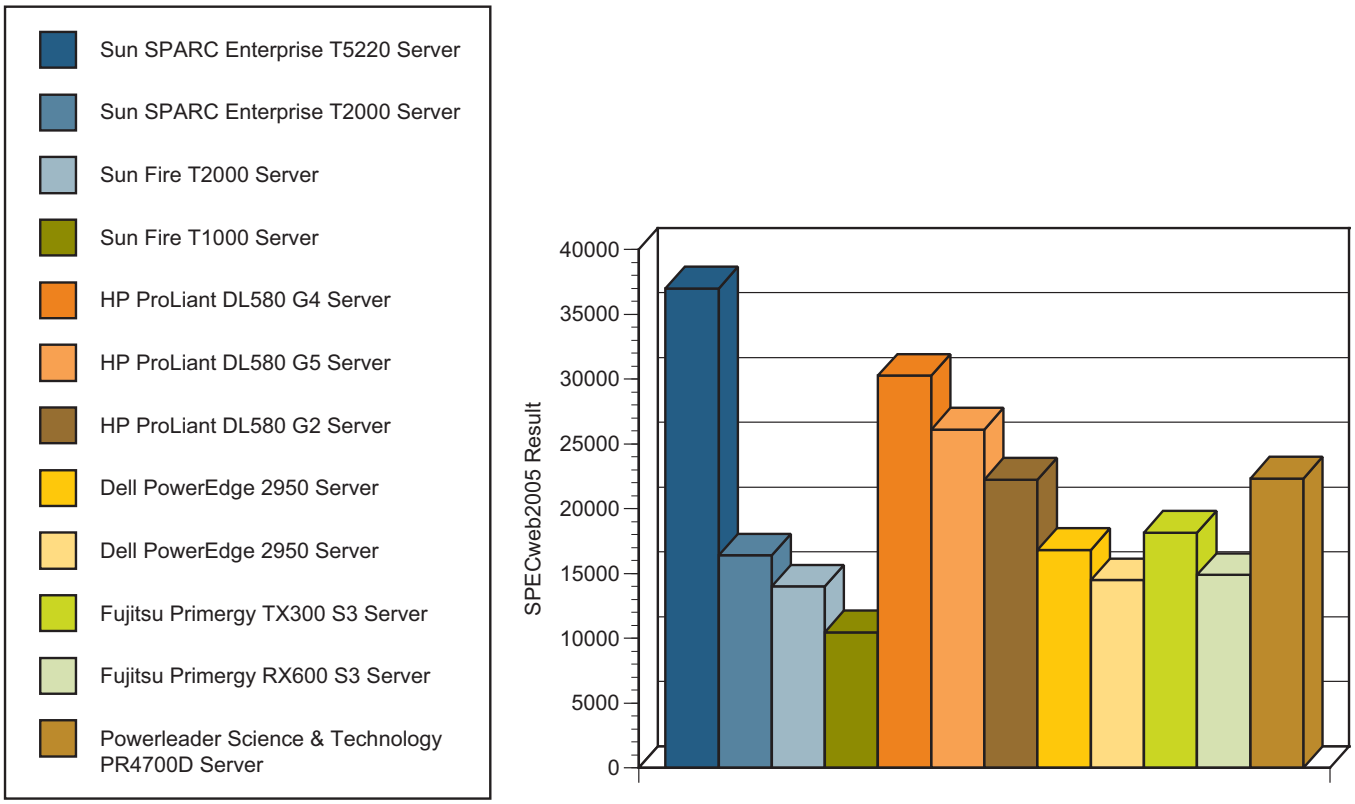


Figure 5. SPECweb2005 results and sub-metrics for different systems as of October 9, 2007

An Archived Oracle Technical Paper

About the Authors

A member of Sun's Performance and Application Engineering (PAE) group, Ning Sun is currently at work on several Web and application server performance projects. Her technical focus centers on the performance of networks, Java technology, and servlet and EJB containers. Ning has extensive background in the development of industry standard benchmarks, including SPECweb, SPECjAppServer, and TPC-W.

Chi-Chang Lin is a member of Sun's Performance and Application Engineering (PAE) group, and has extensive experience with cryptography in embedded systems and telecommunications platforms.

Acknowledgments

The authors thank the following for their advice and contributions to this article:

- Lawrence Spracklen, Architecture Group
- Bhargava Yenduri, Solaris Security Group
- Gary Morton, Solaris Security Group
- Pallab Bhattacharya, Performance and Application Engineering Group

References

Introduction to Cryptography PGP 6.5.1
`http://www.pgpi.org/doc/pgpintro/`

Security APIs, SPIs, and Frameworks for the Solaris OS
`http://developers.sun.com/solaris/articles/security_apis/security_apis.html`

Solaris Security for Developers Guide
`http://docs.sun.com/app/docs/doc/816-4863/`

Solaris Security for Developers Guide, Chapter 8: Introduction to the Solaris Cryptographic Framework
`http://docs.sun.com/app/docs/doc/816-4863/6mb201vgm`

Related References

Handbook of Applied Cryptography, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone

PKCS #11 v2.20: Cryptographic Token Interface Standard, RSA Laboratories, June 28, 2004. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2020/pkcs-11v2-20.pdf`

Standard Performance Evaluation Corporation
`http://www.spec.org`

An Archived Oracle Technical Paper

SPECweb2005 Benchmark
`http://www.spec.org/osg/web2005/`

SPECweb2005 Benchmark Results
`http://www.spec.org/osg/web2005/results/web2005.html`

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at:
`http://www.sun.com/blueprints/online.html`

An Archived Oracle Technical Paper

Appendix A

Introduction to Cryptography and the Secure Sockets Layer

Cryptography

Cryptography is a collection of algorithms that uses mathematics to encrypt and decrypt data. A cryptographic algorithm (cipher), is a set of complex mathematical functions used in the encryption and decryption process. It works in combination with a key (a number, sentence, or word) to encrypt the plain text. The encrypted text is called *ciphertext*. The security strength of the ciphertext is dependent on two factors:

- The strength of the cryptographic algorithm
- How secret the key can be kept

Symmetric-key encryption uses the same key for encryption and decryption. The Data Encryption Standard (DES) is an example of symmetric-key encryption. Sometimes this key is referred to as the Secret Key, and the technique is referred to as Secret Key Encryption. Symmetric-key encryption poses a problem in key distribution between the various trusted parties—the key can be leaked during transmission. The problem of key distribution is addressed by Public Key Cryptography. It is an asymmetric scheme (asymmetric-key encryption) that uses a pair of keys. These keys include a *public key* which encrypts data, and a corresponding *private key* for decryption. Only the public key is published. The private key is kept secret. Anyone with a copy of the public key can encrypt information, however the data can only be decrypted by the owner of the private key.

Since keeping the key secret is no longer an issue, the focus is on making the key pairs as unique as possible. This requires generating large key values. Commonly used keys are 1,024 bits long, and stronger encryption mechanisms use 2,048 bit or 4,096 bit keys. RSA and DSA are two algorithms used in key generation.

Cryptography often involves a one way hashing function that takes variable length input messages of any length and produces a fixed length output. The hash function ensures a different hash value is produced if the information is changed in any way. This process is called the message digest. The message digest is then encrypted with the private key of the sender, and is called a digital signature. The verification of a digital signature involves decrypting the signature using the sender's public key, and then comparing the resultant message digest (hash value) with another hash of the original message which was sent with the digital signature.

An Archived Oracle Technical Paper

Secure Sockets Layer

The Secure Socket Layer (SSL) is a TCP/IP based application of cryptography built around the concept of public key cryptography. Developed by Netscape Communications as an application layer protocol above TCP/IP but just below other application layer protocols like HTTP, SSL has been universally accepted as the protocol for authenticated and encrypted communication between clients and servers. The SSL protocol uses a combination of public key and symmetric key encryption. An SSL session always begins with an exchange of messages called the SSL handshake. The handshake enables the server to authenticate itself with the client using public key techniques. It then enables the client and server to agree on the symmetric keys that will be used for encryption, decryption, and tamper detection during the session that follows. The RSA key exchange algorithm is commonly used with the common cipher suites, including:

- DES and 3DES
- DSA
- MD5, the message digest algorithm developed by Rivest
- RC2 and RC4, Rivest encryption ciphers developed for RSA Data Security
- RSA, a public-key algorithm for both encryption and authentication developed by Rivest, Shamir, and Adleman
- SHA-1, the Secure Hash Algorithm, a hash function used by the U.S. Government

Multiple cryptographic algorithms, and multiple vendors implementing these algorithms, pose an interoperability challenge. Even though vendors may agree on the basic cryptographic techniques, compatibility between implementations is not guaranteed. Interoperability requires strict adherence to agreed upon standards. To help this effort, RSA Laboratories worked with representatives of industry, academia, and government to develop a family of standards called Public Key Cryptography Standards, or PKCS.

Multiple PKCS standards exist. Each standard addresses a set of interoperability issues. PKCS#11 specifies an API, called the cryptographic token interface (Cryptoki), to devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical or abstract view of the device called a cryptographic *token*.

The PKCS11 functionality is split roughly into three parts:

- Administrative operations, including functions like login, session management, and more
- Object management operations, such as create, destroy objects, and more
- Cryptographic operations, including digest, encrypt, and more

An Archived Oracle Technical Paper

The main objects in the PKCS #11 API are slots, tokens, and PKCS11 objects. Token is a general term for devices or placeholders which hold cryptographic information (PKCS11 objects like keys or certificates) and perform cryptographic functions (like digital signatures, random number generation or encryption) after having opened a session. A slot is a container which can potentially hold a token.

PKCS#12 (Personal Information Exchange Syntax Standard) specifies a portable format for storing or transporting a user's private keys, certificates, or miscellaneous secrets.

An Archived Oracle Technical Paper

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN (9786) **Web** sun.com



© 2007 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, Solaris, Sun Blade, Sun BluePrints, Sun Fire, SunDocs, SunSolve, and SunSolve Online are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. SPEC and SPECweb are registered trademarks of Standard Performance Evaluation Corporation. Information subject to change without notice.

Printed in USA 11/07

An Archived Oracle Technical Paper

USING THE CRYPTOGRAPHIC
ACCELERATORS IN THE ULTRASPARC T1
AND T2 PROCESSORS
November 2007

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Copyright © 2011, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd. 1010

Hardware and Software, Engineered to Work Together

An Archived Oracle Technical Paper