nShield Security World

CodeSafe v13.6.11 Developer Guide

30 April 2025

Table of Contents

1. Introduction

CodeSafe is a runtime on the Entrust nShield HSM that allows third-party developers to run their own code within the secure boundary of the module. Using the CodeSafe Developer Kit, developers write their own CodeSafe Apps, cross-compile them and package them to run on the HSM. While on the HSM, the CodeSafe App is segregated from the actual keys loaded onto the module: even the keys the App uses. This means that CodeSafe can be used without affecting the FIPS 140 validation of the module it runs on.

Where the HSMs provide security controls on key usage, CodeSafe provides control over application code. Depending on the runtime used, you're either sending nCore commands to the HSM, or designing your own protocol to send data and commands back and forth.

The CodeSafe™ Developer Kit includes the Secure Execution Engine (SEE) technology. The CodeSafe product comprises a suite of cross-compilers and support tools that allow you to develop SEE machines.

With CodeSafe, you can build and deploy Trusted Agents to perform application-specific security functions on your behalf on unattended servers, or in unprotected environments where the operation of the system is outside of your direct control. Examples of Trusted Agents include digital meters, authentication agents, time-stamps, audit loggers, digital sig nature agents and custom encryption processes.

Traditionally, HSMs have protected cryptographic keys within a defined security boundary; SEE allows you to extend that security boundary to include code that utilizes those pro- tected keys. The code itself can be signed and encrypted to provide additional protection.

This manual applies to both the **nShield Solo XC** and to the **nShield Solo PCIe**.

1.1. Read this guide if ...

Read this guide if you are writing and running SEE applications in C with a SEE-Ready HSM.

This guide:

- Introduces the concept of the Secure Execution Engine (SEE)
- Explains how to use the example SEE machines provided on the installation media
- Describes how to write your own SEE applications in C using the CodeSafe Developer Kit
- Describes how to run your secure SEE applications using a SEE-Ready HSM

- Describes how to obtain export certificates for SEE applications, if required

This guide assumes that you are familiar with the concept of Security World. For information on using keys, including the options and parameters available for the `generatekey` utility, see nShield Security World v13.6.11 Key Management Guide.

This guide assumes that you are familiar with the following documentation:

- The nShield API guides that describe the use of hardware security modules with third-party software products
- The *nCore Developer Tutorial*, which explains how to write applications using a hardware security module
- The *nCore API Documentation* (supplied as HTML), which describes the nCore C API

1.2. Security World Software

The default locations for Security World Software and program data directories on English-language systems are summarized in the following table:

Directory name	Linux default path	Windows environment variable	Windows Server 2016 or later
nShield Installation	`/opt/nfast/`	`NFAST_HOME`	`C:\Program Files\nCipher\nfast`
Key Management Data	`/opt/nfast/kmdata/`	`NFAST_KMDATA`	`C:\ProgramData\nCipher\Key Management Data`
Dynamic Feature Certificates	`/opt/nfast/femcerts/`	`NFAST_CERTDIR`	`C:\ProgramData\nCipher\Feature Certificates`

Static Feature Certificates	`/opt/nfast/kmdata/hsm-ESN/features`		`%NFAST_KMDATA%\hsm-esn\features`
			
			`C:\ProgramData\nCipher\Key Management Data`

Log Files	`/opt/nfast/log`	`NFAST_LOGDIR`	`C:\ProgramData\nCipher\Log Files`
User Log Files	`/home/<user>/nshieldlogs`	`NFAST_USER_LOGDIR`	`C:\Users\<user>\nshieldlogs`
Remote Static Feature Certificates			`%NFAST_KMDATA%\hsm-ESN\features`

Directory name	Linux default path	Windows environment variable	Windows Server 2016 or later
Remote Static Feature Certificates			`%NFAST_KMDATA%\hsm-ESN\features`

Dynamic feature certificates must be stored in the directory stated above. The directory shown for static feature certificates is an example location. You can store those certificates in any directory and provide the appropriate path when using the Feature Enable Tool. However, you must not store static feature certificates in the dynamic features certificates directory.

The instructions in this guide refer to the locations of the software installation and program data directories as follows:

- By name (for example, Key Management Data).
- **Linux**: By absolute path (for example, `/opt/nfast/kmdata`).
- **Windows**: By nShield environment variable names enclosed with percent signs (for example, `%NFAST_KMDATA%`).

`NFAST_KMDATA` cannot be a symbolic link.

If the software has been installed into a non-default location:

- **Linux**: Create a symbolic link from `/opt/nfast/` to the directory where the software is actually installed.
- **Windows**: Ensure that the associated nShield environment variables are re-set with the correct paths for your installation. For more information about creating symbolic links, see your operating system's documentation.

Windows only

By default, the Windows `C:\ProgramData\` directory is a hidden directory. To see this directory and its contents, you must enable the display of hidden files and directories in the view settings of the Folder Options.

The absolute paths to the Security World Software installation directory and program data directories are stored in the indicated nShield environment variables at the time of installation If you are unsure of the location of any of these directories, check the path set in the environment variable.

> With previous versions of Security World Software, the Key Management Data directory was located by default in `C:\nfast\kmdata`. The Feature Certificates directory was located by default in `C:\nfast\fem`, and the Log Files directory was located by default in `C:\nfast|log`.

1.2.1. Utility help options

Unless noted, all the executable utilities provided in the `bin` subdirectory of your nShield installation have the following standard help options:

- `-h|--help` displays help for the utility
- `-v|--version` displays the version number of the utility
- `-u|--usage` displays a brief usage summary for the utility.

1.3. Requirements

To write and run a SEE C application on the HSM, you need:

- A SEE-Ready hardware security module

> To determine whether your HSM is SEE-Ready, refer to the product data sheet for your HSM.

> Encrypted SEE machines are not currently supported for use with nShield Connects. When the SEEMachine binary is installed on the Connect itself for automated loading at boot, the SEE Confidentiality key is not available. However, when a client host loads a SEEMachine, it has access to the SEE Confidentiality key and can cause the binary to be decrypted. In this scenario, the Connect works fine with encrypted SEEMachine binaries.

- A Feature Enable smart card for activating the SEE capabilities of your HSM
- The CodeSafe Developer Kit (supplied on this installation media)
- An appropriate GCC compiler (supplied on this installation media) for the target HSM.

You must have installed your SEE-Ready HSM and the necessary Security World for nShield for the CodeSafe Developer Kit. You must install at least the following software component bundles included on the installation media:

- `hwsp Hardware Support`

- `ctls Core Tools`
- `csd CodeSafe Developer`
- `gccsrc Prebuilt PowerPC GCC for CodeSafe/C`

When you have installed and configured your SEE-Ready HSM, to make full use of SEE, you must create a Security World by using one of the following tools:

- new-world
- the front panel (only on network-attached HSMs).

1.4. Further information

This guide forms one part of the information and support provided by Entrust.

The *nCore API Documentation* is supplied as HTML files installed in the following locations:

- API reference for host:
  - **Linux**: `/opt/nfast/document/ncore/html/index.html`
  - **Windows**: `%NFAST_HOME%\document\ncore\html\index.html`
- API reference for SEE:
  - **Linux**: `/opt/nfast/document/csddoc/html/index.html`
  - **Windows**: `%NFAST_HOME%\document\csddoc\html\index.html`

We recommend that you monitor the Announcements & Security Notices section on Entrust nShield Support, https://nshieldsupport.entrust.com, where any announcement of Security advisories will be made.

1.5. Security advisories

If Entrust becomes aware of a security issue affecting nShield HSMs, Entrust will publish a security advisory to customers. The security advisory will describe the issue and provide rec ommended actions. In some circumstances the advisory may recommend you upgrade the nShield firmware and or image file. In this situation you will need to re-present a quorum of administrator smart cards to the HSM to reload a Security World. As such, deployment and maintenance of your HSMs should consider the procedures and actions required to upgrade devices in the field.

The Remote Administration feature supports remote firmware upgrade of nShield HSMs, and remote ACS card presentation.

We recommend that you monitor the Announcements & Security Notices section on Entrust nShield, https://nshieldsupport.entrust.com, where any announcement of nShield Security Advisories will be made.

1.6. Contacting Entrust nShield Support

To obtain support for your product, contact Entrust nShield Support, https://nshieldsupport.entrust.com.

2. About the Secure Execution Engine SEE

The *Secure Execution Engine* (SEE) enables application code to run within the secure environment of a SEE-Ready HSM.

To use SEE, you must order and enable it first, see Optional features. You must order the developer and user environments separately. SEE machines cannot be loaded on HSMs on which SEE is not enabled.

The CodeSafe Developer Kit includes the following:

- The CodeSafe Developer Libraries
- A built GCC compiler, plus source and makefile to customize your own version, if required
- The CodeSafe Utilities (described in Utilities):
  - tct2 (the Trusted Code Tool)
  - elftool
  - loadsee-setup
  - loadmache (for use with `SEElib`)
  - hsc_loadseemachine
  - `seessl-migrate.py`
  - a set of host utilities (for use with the Solo XC `glibc`-based SEE machines) that enable the standard IO and socket connections: see-sock-serv, see-stdioe-serv, see-stdioesock-serv, see-stdoe-serv:
    - `see-sock-serv`
    - `see-stdoe-serv`
    - `see-stdioe-serv`
    - `see-stdioesock-serv`.

2.1. Why use the Secure Execution Engine?

The main uses of cryptography are:

- Integrity
- Confidentiality
- Authentication

Using an HSM to protect your cryptographic keys provides all these advantages. Your keys are only ever available in unencrypted form when they are loaded into the HSM: when key

blobs are stored on the host, their integrity is protected by a Message Authentication Code (MAC). Access to the keys is controlled by using a Security World or an Operator Card Set (OCS).

However, traditionally, the code that uses the keys remains on the server. This means that the code is open to attack. It is possible that the code could be modified in such a way as to leak important information or compromise your business rules. For example, it could fail to enforce such rules as “the books must balance” or “traders shall balance their positions by the close of trading”.

By implementing a solution with the SEE, you not only protect your cryptographic keys but also extend the security boundary to include your security critical code and data.

Using the techniques of code signing, data wrapping, and secure storage, the SEE enables you to maintain the confidentiality and integrity of application code and data and to bind them together so that only code in which you have confidence has access to confidential data.

2.1.1. Code integrity

In many secure applications, the primary concern is for the code to execute the correct sequence of operations and to not do anything else, such as leak information or key data. You can use the supplied Trusted Code Tool (`tct2`) to sign the HSM-side code and initializa tion data (if required) that make up a SEE machine. Application authors can use signatures to delegate authority to use key material and other resources.

2.1.2. Code confidentiality

When you use the SEE, the code that runs on an HSM can be stored in an encrypted for- mat. The encryption key can be either a Triple Data Encryption Standard (Triple DES) or Advanced Encryption Standard (AES) key protected by either a Security World or an OCS.

Encrypted SEE machines are not currently supported for use with nShield Connects. When the SEEMachine binary is installed on the Con- nect itself for automated loading at boot, the SEE Confidentiality key is not available. However, when a client host loads a SEEMachine, it has access to the SEE Confidentiality key and can cause the binary to be decrypted. In this scenario, the Connect works fine with encrypted SEEMachine binaries.

The Access Control List (ACL) entry, `UseAsLoaderKey`, enables a key to be used to decrypt

SEE objects on the HSM but that does not allow you to use it for standard decryption where the answer is returned to the host. This ensures that the code itself is not available "in the clear" outside of the HSM/SEE and; therefore, that any intellectual property embodied in the code is protected.

To load encrypted code, the user must first load the encryption key. Therefore, if the encryption key is protected by an OCS, only users with sufficient smart cards from that OCS can load the code. Because this SEE confidentiality key does not have decryption permissions (only the `UseAsLoaderKey` ACL entry), from a security standpoint, it is not essential that it be protected by an OCS.

HSM-protected SEE confidentiality keys can be useful in situations where the server or HSM is unexpectedly reset, because, in such a case, the SEE machine can then be reloaded without user intervention.

2.1.3. Data confidentiality

There are two main issues regarding data confidentiality:

- Transient confidentiality of data in the running system
- Long-term confidentiality of data when the code is not loaded.

The SEE protects the program's information in the running system by enabling the programmer to determine the interface by which data can come in and out of the system and then rigorously enforce that interface.

Long-term confidentiality is preserved by using the non-volatile memory on the HSM. The SEE program can access this storage by using nCore API commands. Small quantities of highly sensitive information can be stored directly in the nonvolatile random access memory (NVRAM). When the amount of information to be stored exceeds the capacity of the NVRAM, data can be stored in an encrypted blob with a much smaller key stored in the NVRAM. This functionality allows the amount of secure storage to be limited only by the capacity of the host. For more information, see the *nCore Developer Tutorial*.

2.1.4. Data integrity

Confidential data is of little use if it can be changed by an attacker. Data stored in the HSM's NVRAM could only be altered if the Access Control List (ACL) were to allow this to happen or if the physical security of the HSM were compromised. When a large volume of data is made into a blob, a hash of that blob can be stored in the NVRAM so that changes can be detected.

Another option for maintaining data that is not likely to change (such as root CA keys) is to place it in the **application initialization space** and then use code integrity techniques to pro tect the **application initialization space**.

2.1.5. Authentication and access control

A key feature of the SEE is the way that it can tie the integrity of the code to access control of the resources that the code uses.

The key-management architecture controls access to objects such as keys by means of ACLs. These lists specify sets of operations and verification keys that are used to check the credentials authorizing these operations.

With SEE, you can create keys that can only be used to encrypt or sign SEE machines (the SEE HSM-side code and, if required, its `userdata`). Encrypted application code is effectively bound to the encryption key, thereby ensuring that it can only be loaded onto an HSM on which you have already loaded the key. This functionality effectively gives you OCS protection on application code.

Encrypted SEE machines are not currently supported for use with nShield Connects. When the SEEMachine binary is installed on the Connect itself for automated loading at boot, the SEE Confidentiality key is not available. However, when a client host loads a SEEMachine, it has access to the SEE Confidentiality key and can cause the binary to be decrypted. In this scenario, the Connect works fine with encrypted SEEMachine binaries.

SEE also extends the authorization credentials to include signatures on code. This simple extension turns out to be very powerful. When a body of code issues a command to use a resource that is controlled by an ACL, it may present a certificate indicating that the signatures on the code should be examined by the ACL checking system. If the signature on the code verifies with one of the keys listed in the ACL, the operations delegated to that key can be carried out in that command.

Therefore, this extension of the authorization credentials means that you can create keys that can only be used by the SEE-resident code. These keys can be protected by the Security World or by OCSs.

The SEE code has access to the HSM's NVRAM. Files stored in the HSM's non-volatile mem ory also have ACLs. These ACLs describe not only who can access the file but what changes can be made to the file. For example, this feature enables you to create secure counters that you know can never be zeroed or that you know can be zeroed only by a

trusted application running in the SEE.

2.2. How SEE works

A hardware security module maintains strict separation between the nShield core functions and the user code.

The application starts with the code for a SEE machine stored in a file on the host. A SEE machine is a binary executable of a type appropriate for the HSM. It communicates with the nShield Solo XC core by means of the **interprocess communication** (**IPC**).

Applications may be written in C and compiled to form the SEE machine itself. Alternatively, the SEE machine may consist of a language interpreter and the HSM code supplied as a script or byte code by means of `userdata`. For more information, see SEE and userdata.

If a separate host-side program is required, you can write the host-side code in C, using the nCore API. Alternatively, you can use the language of your choice. Example utilities written in Java are provided in the component `jhsee` in `/opt/nfast/java/examples` (**Linux**) or `%NFAST_HOME%\java\examples` (**Windows**).

These example utilities provide equivalent functionality to the C examples of similar names. You can adapt them as required. See the supplied Javadocs for full information about the Java example utilities.

The SEE machine can be signed, encrypted, or both, with the `Trusted Code Tool` (`tct2`). For more information about this command-line utility, see Utilities.

Encrypted SEE machines are not currently supported for use with nShield Connects. When the SEEMachine binary is installed on the Connect itself for automated loading at boot, the SEE Confidentiality key is not available. However, when a client host loads a SEEMachine, it has access to the SEE Confidentiality key and can cause the binary to be decrypted. In this scenario, the Connect works fine with encrypted SEEMachine binaries.

The first step is to load the SEE machine onto the HSM. The hardserver software, supplied on this installation media, automatically loads the SEE machine whenever the HSM is reset, provided that:

- The HSM is SEE-Ready

To determine whether your HSM is SEE-Ready, refer to the product data sheet for your HSM.

- The HSM sets the enquiry level 4 `HasSEE` flag
- A suitable machine image file is configured
- The `load_seemachine` section of the configuration file is configured to enable the loading of SEE machines on startup.

You can perform this configuration with the `loadsee-setup` command-line utility. See Utilities.

For development purposes, you can also load SEE machines manually by running the `loadmache` command-line utility or, optionally, you can load SEE machines that require support from a host-side `see-*-serv` utility by specifying the `-M` option when you run the utility. See Utilities.

2.2.1. Code specifics

To use the functions provided by the SEE machine, the host application creates a `SEE World`, supplying the initialization data, which includes the HSM resident portion of the application code, initialization flags and any other SEE World initialization information required. The functions provided by the HSM-resident code can then be accessed by the SEE machine on command from the host-side portion of the application. The SEE World is a private work space and has a handle, an `M_KeyID`. As with other identifiers, this handle is associated with a `ClientID`. A host application can only access a `SEEWorld` on the connection that created the `SEEWorld` or on connections that have the same `ClientID`.

The `CreateSEEWorld` command takes a byte block called the `SEE user data`. This block can be used to pass initialization data when a SEE machine is started. This file also carries the signatures for the `SEEWorld`.

Refer to the *nCore CodeSafe API Documentation* for detailed information about the `CreateSEEWorld` command.

2.2.2. Security

When the SEE machine has been initialized, the host application can call the functions that the SEE machine provides. These calls are sent using the nCore API command `SEEJob`.

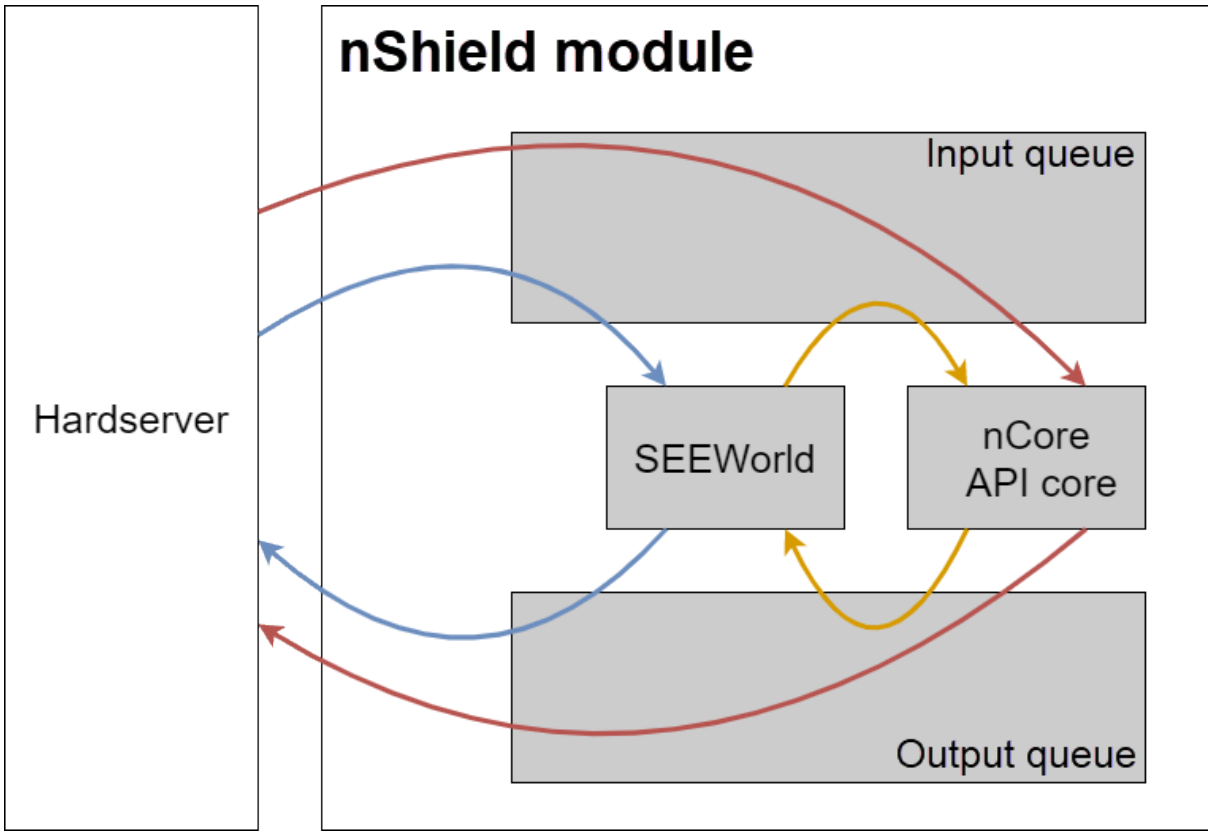
For example, if you write code to implement a custom algorithm, the host application no longer calls the nCore API `Encrypt` command. Instead, it calls the encrypt function of the SEE machine. The algorithm in the SEE machine then asks the core for the key, uses the key to encrypt the message, and returns the result. This is explained in detail for the Solo XC in Designing SEE machines and SEE-ready HSMs.

The SEE machine can then make calls into the nShield core with the standard nCore API. The replies are returned directly to the SEE machine without ever leaving the protection of the HSM.

The SEE machine can access keys, or other objects that are protected by the HSM, only by making nCore API calls to the nShield core. HSM-side SEE code has the same privileges and access to the cryptographic functionality of the HSM as that given to the host-side programs using the nCore API. However, it is possible to create SEE application keys that can be used only by particular SEE applications and not by the host.

2.2.3. Internals

CodeSafe uses two command queues; The following diagram gives an overview of how they function. The hardserver sends commands to the input queue. The input queue looks at the commands and directs them to either the nCore API core or to the `SEEWorld`.



In this release you can only create a single `SEEWorld` for each HSM at any one time.

The nCore API core takes commands from the input queue, processes them in turn, and

places them on the output queue. These commands may have come from the server or from the SEEWorld.

The output queue receives the completed jobs from the core. It determines whether the command was issued by the SEEWorld or the hardserver and sends the result to the appropriate place.

While any command sent to the SEEWorld may cause a number of calls to the nCore API core (and these calls circulate within the HSM), a given command only ever produces a single reply that is returned to the server. After the SEEWorld has completed the job, it returns a reply. The core returns this reply to the hardserver and on to the application; this is the reply to the SEEJob command, handled in exactly the same manner as for any other nCore API command.

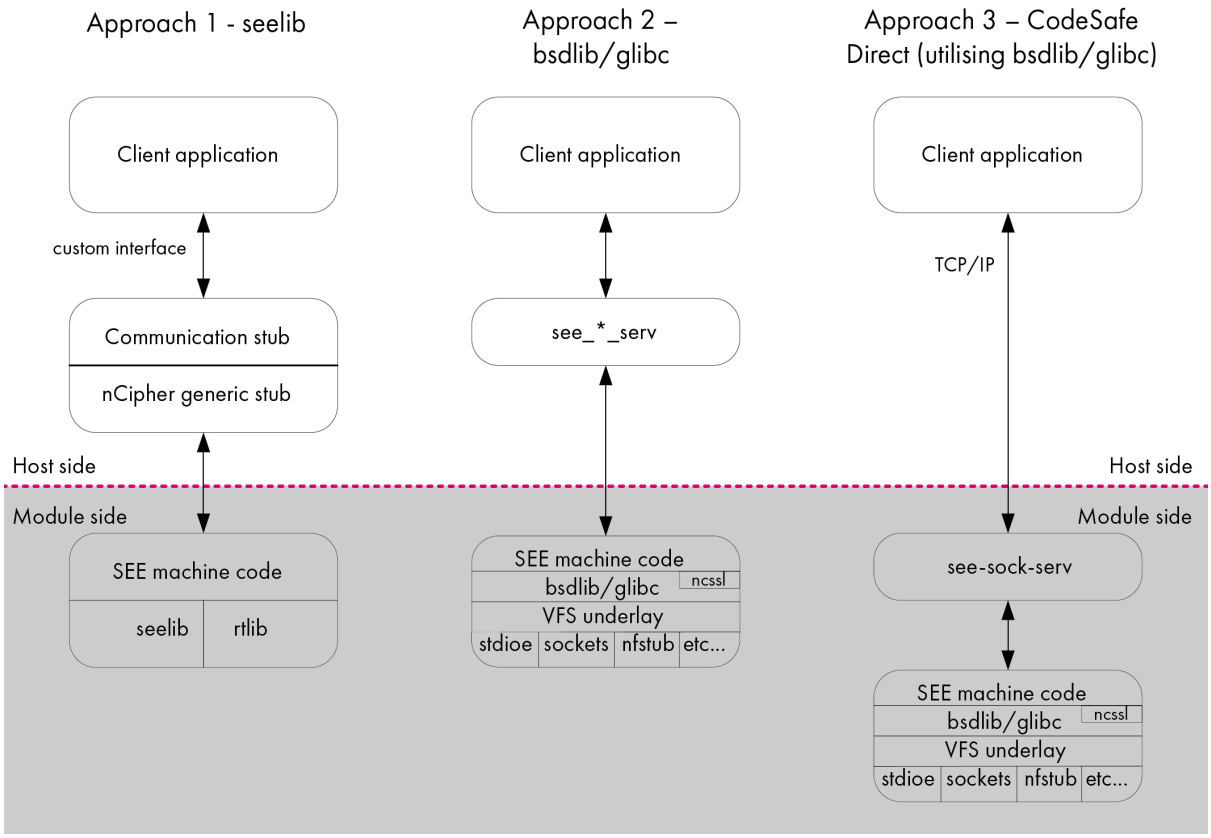
The SEEJob reply is returned with Status_OK provided that the SEE machine returns a reply to the nShield core. The return of this kind of reply does not mean that the command itself was completed successfully in the SEE machine, only that communication between the core and the SEE machine was completed successfully. The SEE machine returns its own errors (if any) in the reply.

The application running in the SEEWorld does not have direct access to the user interface. Therefore, all interaction with the user must be performed by the host application. In some cases, especially when loading tokens that are protected by multiple smart cards, it can be useful to have the host application load an object and then pass control to the application in the Status_OK. You cannot pass the ObjectID because this is specific to the ClientID. Therefore, to pass control to the application in the Status_OK, you must use key tickets.

Key tickets were introduced to the nCore API specifically for SEE, although they can also be used to pass keys between different clients on the host. The client (or SEE application) that creates a key asks for a ticket for the key. It passes the ticket to the other client, which redeems the ticket for an ObjectID. There is only ever one copy of the object, and all commands have to comply with the ACL.

2.3. SEE system architecture

There are different architectural strategies that you can use when designing a CodeSafe SEE system, distinguished by the library they utilize:



Before designing your CodeSafe SEE system, decide which architecture best suits your requirements:

- `glibc`: This architecture allows the use of TCP sockets and a high performance GNU C library in CodeSafe. This makes it possible to communicate with a SEE machine using a generic approach.

  `glibc` can only be used if you are using an nShield Solo XC module and supports ISO C, POSIX, and System V standards.

  A design using this architecture is well suited for SEE machines that implement applica tions such as Web servers and proxies.

  If you are designing a CodeSafe Direct system, you must use the `glibc` architecture. The `SEElib` library is *not* supported for use with CodeSafe Direct.

  If you are designing a CodeSafe SEE system using the `glibc` library, you can use headers as normal for a Unix-based system (for exam ple, `stdio.h`, `stdlib.h`, `pthread.h`).

- `SEElib`: A design using this legacy architecture is well suited to protecting custom cryp tography within a SEE machine. The `A3A8` example program provides a simple demon-

stration of how to achieve this; see Designing SEE machines and SEE-ready HSMs for additional information.

If you are designing a CodeSafe SEE system using the `SEElib` library, you can use the header file `seelib.h`, which contains wrapper functions for the software interrupts, in addition to a limited subset of the standard C library. See SEElib functions for additional information.

Unless you have a specific reason to use the `SEElib` architecture, Entrust recommend using the `glibc` architecture, as it provides a more familiar standards-based programming environment using standard socket and standard IO interfaces. Note that `SEElib` typically requires additional work on the host application to interface to the SEE code. This is not required when using the standards-based `glibc` approach.

2.4. SEE and userdata

2.4.1. What is userdata?

A `userdata` file can contain any data that is useful to the SEE machine. For example, you can use a CPIO archive to supply many different data files in a single directory structure (examples are provided in Designing SEE machines and SEE-ready HSMs.

All SEE machines built with `glibc` must be provided with a valid ASCII-format CPIO archive. This archive forms the base of the file system available to your SEE machine. Even if your SEE machine does not use this file system, you must still create and supply it with dummy `userdata` as a place-holder.

2.4.2. Creating userdata suitable for loading into the HSM

You can create a `userdata` file suitable for loading into the HSM by turning it into a SAR file with the `tct2` command-line utility. Signing the `userdata` file in this way offers improved security.

2.5. SEE and Security Worlds

Within a Security World, the following actions may be configured to require authorization from the nShield Security Officer Key (K_{NSO}) , or a key with authority delegated from the K_{NSO}:

- Allocation and forced freeing of nonvolatile memory
- Setting the real-time clock
- Enabling the run-time debugging options.

Each of these features can be enabled individually.

At Security World creation time, certificates may be created delegating authority from K_{NSO} to keys protected by logical tokens which are split amongst the Administrator Card Set (ACS) in the usual way, but may require a different K/N threshold to reassemble. For example, you may wish to require that only one of five Administrator Cards be presented to set the real-time clock on an HSM, but three of them to replace the ACS.

The tools that create these certificates are:

- KeySafe (version 2 and later)
- **Windows**: The nShield CSP Wizard
- new-world
- the front panel (only on network-attached HSMs)

A Security World created using some older tools does not have any of these delegation certificates to support nonvolatile memory and real-time clock operations or to allow debugging of SEE applications. Therefore, such operations would require full K_{NSO} authorization.

To sign or encrypt the HSM-side code, the signing and encryption keys must belong to the Security World to which the HSM belongs.

To test code outside a Security World, you can use the `initunit` command-line utility to remove the HSM from the Security World. In this case you cannot sign or encrypt your code, and the code cannot access keys protected by the Security World.

If you use the `initunit` command-line utility to initialize the HSM, any user can set the clock and create or free NVRAM files. This means that any user can free an existing file and allocate another file with the same name but with different contents or with a different ACL. Most security policies forbid this.

3. Designing SEE machines and SEE-ready HSMs

This manual addresses SEE for the Solo XC and Connect XC.

For Solo XC, see Writing SEE machines - Solo XC

3.1. Writing SEE machines - Solo XC

This chapter describes how to write a SEE machine for use on SEE-Ready HSMs.

An SEE machine is an executable binary file of a type appropriate for the HSM that communicates with the nShield core (which runs in kernel mode) using a defined set of software interrupts. These interrupts, and their wrapper functions, provide a run-time environment that includes memory and thread management as well as an interface for accepting and returning jobs and calling nCore API commands.

C source code is compiled using one of the GCC cross-compilers supplied with the CodeSafe Developer Kit. For details of required compiler options; see Example SEE machines and the makefiles supplied with the examples.

The compiled code can then be signed, packed, and encrypted by using the Trusted Code Tool (`tct2` utility) to produce a secure archive; see Utilities.

In CodeSafe versions prior to 13.3, the Solo XC only supports SEE machines smaller than 70 MB. From 13.3 onwards, the Solo XC can support SEE machines up to 800 MB.

3.1.1. Designing for the glibc architecture

The GNU C library `glibc` is supplied together with `libpthreads`, `librt` and a system call underlay for use with CodeSafe SEE development.

A rich set of C function calls is available to use in SEE machine development. Native support for Unix-based system calls is provided, only restricted by an allowlist of the system calls (Allowlist for SEE machines) allowed in the SEE environment.

A subset of the Unix-based system calls, implemented in terms of the inter-process communication interface (IPC), allows access to the cryptographic HSM kernel. The provided system calls include a virtual file system and associated set of input and output devices with which you interact in the standard manner.

The virtual file system is supported as an extension to the file system.

Also provided are some link-time plug-ins that extend the virtual file system to provide addi tional capabilities:

- `hoststdioe.o`: `stdin`, `stdout`, and `stderr` facility hooks; `seestream_stdio(7see)`
- `hoststdoe.o`: `stdout` and `stderr` facility hooks; `seestream_stdio(7see)`
- `hostinetsocks.o`: TCP socket facility hooks; `seestream_inet(7see)`
- `hoststdioeinetsocks.o`: TCP socket facility and `stdin`, `stdout`, and `stderr` facility hooks; `seestream_inet(7see)seestream_stdio(7see)`

The link-time plug-in `vulnerability.o` is provided for the purposes of debugging (see *Vulnerability test harness*). Entrust recommends that you do not link `vulnerability.o` into a production SEE machine.

3.1.2. Designing for the SEElib architecture

This section describes how to design SEE machines using the `SEElib` architecture. This kind of architecture requires host-side software to create the SEE World and communicate with the HSM.

To start the SEE machine running with a particular SEE `userdata`, the host application calls the nCore API command `CreateSEEWorld`. This command creates a SEE World using data previously loaded into the HSM with the `LoadBuffer` command from a buffer created with the `CreateBuffer` command. See the *nCore API Documentation* (supplied as HTML) for information about the nCore API commands.

You can also use or adapt the supplied example Java class `SEEWorld` to initialize the SEE machine.

When the host application calls `CreateSEEWorld`, the HSM allocates memory for the SEE World and sets up its input and output job queues. It then runs the SEE machine's `main()` function.

The SEE machine's `main()` function must:

- Call `SEElib_init()` before any other SEE library function to initialize the SEE library and to check that the HSM is running the expected version of the library
- If the machine accepts `userdata`:
  - call `SEElib_GetUserDataLen` to determine the length of the byte block that was passed with `CreateSEEWorld`
  - call `SEElib_ReadUserData` to load the byte block

    - determine whether the byte block is valid
    - initialize any required structures
- Start at least one thread which receives and processes commands (this thread must call `SEElib_AwaitJob`)

The `SEElib_StartProcessorThreads` function can be used for this purpose.

- Call `SEElib_InitComplete` and return a status.

The status passed to `SEElib_InitComplete` is returned to the calling application in the reply to `CreateSEEWorld` The application can determine the status values, with one exception: if the machine fails before calling `SEElib_InitComplete()`, the `CreateSEEWorld` command returns a value of `1` (`SEEInitStatus_MachineFailed`) in this field. You should therefore avoid choosing the value `1` to indicate successful initialization.

When the application receives the reply to `CreateSEEWorld` with `Status_OK` and an acceptable `initstatus`, it can start to submit jobs with the nCore API command `SEEJob`.

You can also use or adapt the supplied example Java class `SEEJob` to submit jobs to the SEE machine.

The `SEEJob` command takes a byte block, which is passed to the `SEElib_AwaitJob` function without being interpreted in any way. It is up to the host application to assemble this byte block and the SEE machine to interpret it.

After the job has been processed, assemble the reply into a byte block and call `SEElib_ReturnJob` to return it using the nShield core.

The nShield core assembles this byte block into a reply and returns it to the host application. Provided that the job is returned before the command times out, the reply has the status `OK`. The SEE machine must include any necessary status information within the byte block it returns. The calling application must remember to check this status as well as the status of the `SEEJob` nCore API function and the transport call, for example `NFastApp_Transact()`.

The SEE machine can call nCore API functions with `SEElib_Transact` or `SEElib_MarshalSendCommand` and `SEElib_GetUnmarshalResponse`. It may submit these as part of its initialization, before it calls `SEElib_InitComplete()`. However, if it does not call `SEElib_InitComplete()` within 30 seconds of start-up, the `CreateSEEWorld` command returns `SEEInitStatus_MachineFailed`. For this reason, you should not perform (for example) lengthy key generation operations during initialization.

`SEElib_Transact` has syntax equivalent to the `NFastApp_Transact` function in the C generic

stub. It takes a command structure and returns a reply structure. `SEElib_MarshalSendCommand` takes a command structure and submits it. `SEElib_GetUnmarshalResponse` reads a response from a buffer and returns a reply structure.

`SEElib_StartTransactListener` must be called successfully before you use `SEElib_Transact` to communicate with the nShield core.

3.1.3. SEE machines for new algorithms

In addition to being able to perform basic cryptographic operations, any SEE machine that implements an algorithm must also be able to:

- Generate keys
- Import keys
- Store keys as key blobs.

The SEE machine can use the nCore API functions `GenerateRandom` and `GeneratePrime` to acquire random numbers and random prime numbers from the HSM's hardware random number generator.

The SEE machine can perform its own multiprecision arithmetic. Otherwise, it can use the nCore API `BignumOp` command to perform multiprecision arithmetic and the `ModExp` and `ModExpCrt` commands to perform modular exponentiation.

If you are using keys as session keys, there is no requirement for them ever to be placed in the nShield core. The only time that you need to transfer a key to the core is if you need to create a key blob for long-term storage. However, if you need to keep track of several keys, you may want to make use of the nShield core's object store rather than having to create a similar structure in your own code.

For an example of how See machines can implement a non-standard algorithm, see A3A8 example.

3.1.3.1. Key type

The SEE machine stores keys using the `random` key type. This is a plain byte block with no structure.

If the key contains several values, for example, exponent and modulus, the SEE machine must implement its own routines for marshalling and unmarshalling the byte block into the correct structure.

SEE machines using standard algorithms do not use the `random` key

> type. Instead, they use standard nCore key types.

3.1.3.2. ACL

The ACL needs to be constructed so that the SEE machine and only the SEE machine can access the key. To transfer a key from the nShield core to the SEE machine, the key must have the `ExportAsPlain` flag set in its ACL. The permission group with `ExportAsPlain` must be protected by a certifier so that this operation can only be performed by the SEE machine.

Although one obvious solution is to use the key that was used to sign the SEE machine, K_{Integ}, as the certifier, using K_{Integ} in this way means that whoever signed the SEE machine could potentially access any key for this algorithm. A better solution is to add an extra signa ture to the SEE machine by using a second key, K_{Auth} The K_{Integ} signature proves that the code has not changed since it was signed. The K_{Auth} signature is then used to control access to keys.

You can use the `generatekey` command-line utility to generates keys for use as K_{Auth} and K_{Integ} by specifying the `seeinteg` application as a key generation parameter.

The ACL must also have the correct `MakeBlob` permissions. If you want to use the standard Security World tools for key management and recovery, the host application can use these tools to create the ACL.

> SEE machines using standard algorithms generally do not need to get the key as plain text in the SEE machine.

3.1.3.3. Storage

For long-term storage, the key needs to be encapsulated in a key blob that is protected by the Security World or an OCS. To provide OCS replacement and recovery, you may also require additional key blobs protected by other card sets.

You could write a function where your SEE machine calls `MakeBlob` and returns the blob to the host. Alternatively, you could write a method that returns a key ticket and have the host application create the key blobs.

If you are using a Security World, the host application can use `nfkm` library calls to create and store the key blobs.

3.1.3.4. Loading stored keys

In general, it is easier for the host application to manage tokens, because it has direct access to the user interface and can prompt the user to insert cards and enter passphrases.

When the token has been loaded, the host application can load the key and pass a key ticket to the SEE machine. The SEE machine can then redeem the key ticket for a `KeyID` and use this to access the key. If you have several keys that are protected by a token, it usually makes sense to pass a ticket for the `KeyID` of the logical token, rather than passing tickets for each key.

You should also pass in a ticket for the logical token if the host application that loads the token exits afterwards. When it exits, it destroys the logical token's ID, which invalidates all loaded keys that were using it. Passing the logical token's ID in to the SEE machine prevents its destruction when the application exits.

3.1.3.5. C run-time library

Entrust supplies a customized version of the GNU C (glibc) library for Solo XC SEE machines. Common features such as threading and mutexes are provided by `glibc`.

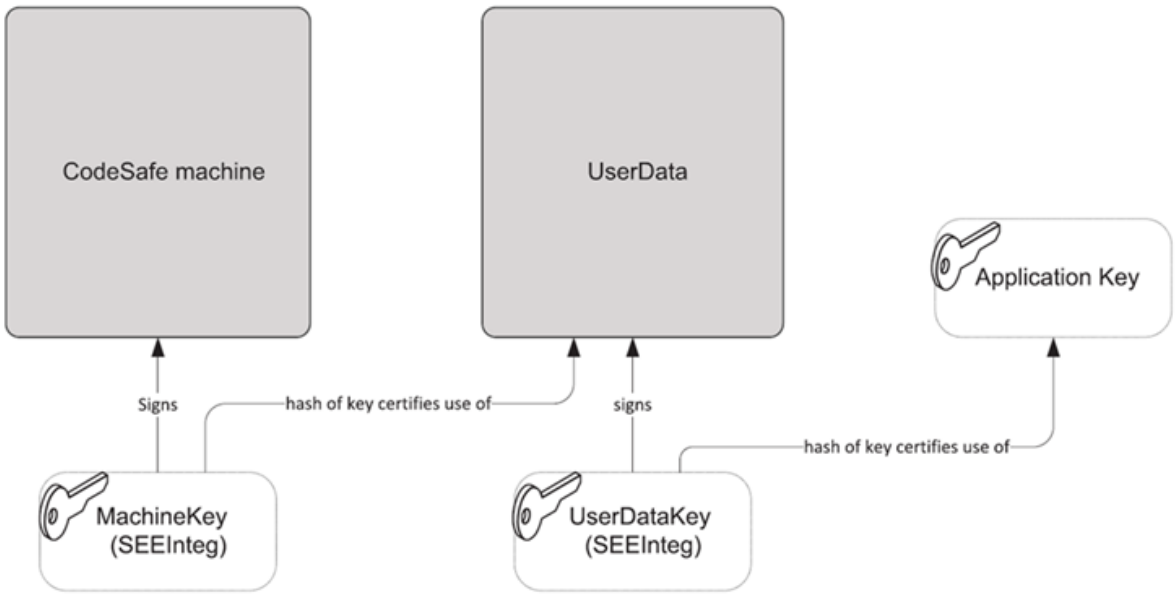
See SEElib functions for reference information about `glibc` functions.

3.1.4. Signing userdata for additional security

Signing `userdata` files can help increase the security of CodeSafe SEE applications. Both types of SEE machine architecture, using glibc and using SEElib, can take advantage of the security benefits offered by signing `userdata` files.

For example, if your SEE machine is intended to perform some cryptography functions using a given key, it would be advantageous to prevent that key from being accessed by any unauthorized SEE machines. This can be achieved by signing the `userdata` file for your SEE machine.

The following figure provides an overview diagram of the process of signing a SEE machine's `userdata` file.



The following sequence, in which an original SEE machine is represented by `machine.elf` and an original `userdata` file is represented by `userdata.bin`, demonstrates the process of signing a SEE machine's `userdata` file:

1. Create the key $K_{seemach}$ of type `seeinteg` to sign the SEE machine by running a command similar to:

```
generatekey seeinteg plainname=seemach ...
```

2. Create the key $K_{userdata}$ of type `seeinteg` to sign the `userdata` by running a command similar to:

```
generatekey seeinteg plainname=userdata ...
```

3. Run the `generatekey` command-line utility to create a key K_{crypto} (the key with which your SEE machine is to perform its cryptography functions), specifying $K_{userdata}$ for its `seeintegname`:

```
generatekey simple plainname=crypto --seeintegname=userdata ...
```

This example assumes K_{crypto} is being created as a Triple DES key.

4. Run the `tct2` command-line utility to sign the `userdata` file for your SEE machine with the key $K_{userdata}$, specifying $K_{seemach}$ as the SEE machine key:

```
tct2 --sign --key=userdata --machine-key-ident=seemach --infile=userdata.bin --outfile=userdata.sar
```

For information about the `tct2` command-line utility, see tct2.

5. Run the `tct2` command-line utility to sign the SEE machine with the key $K_{seemach}$:

```
tct2 --sign --key=seemach -- machine-type=PowerPCELF --is-machine --infile=machine.elf
--outfile=machine.sar
```

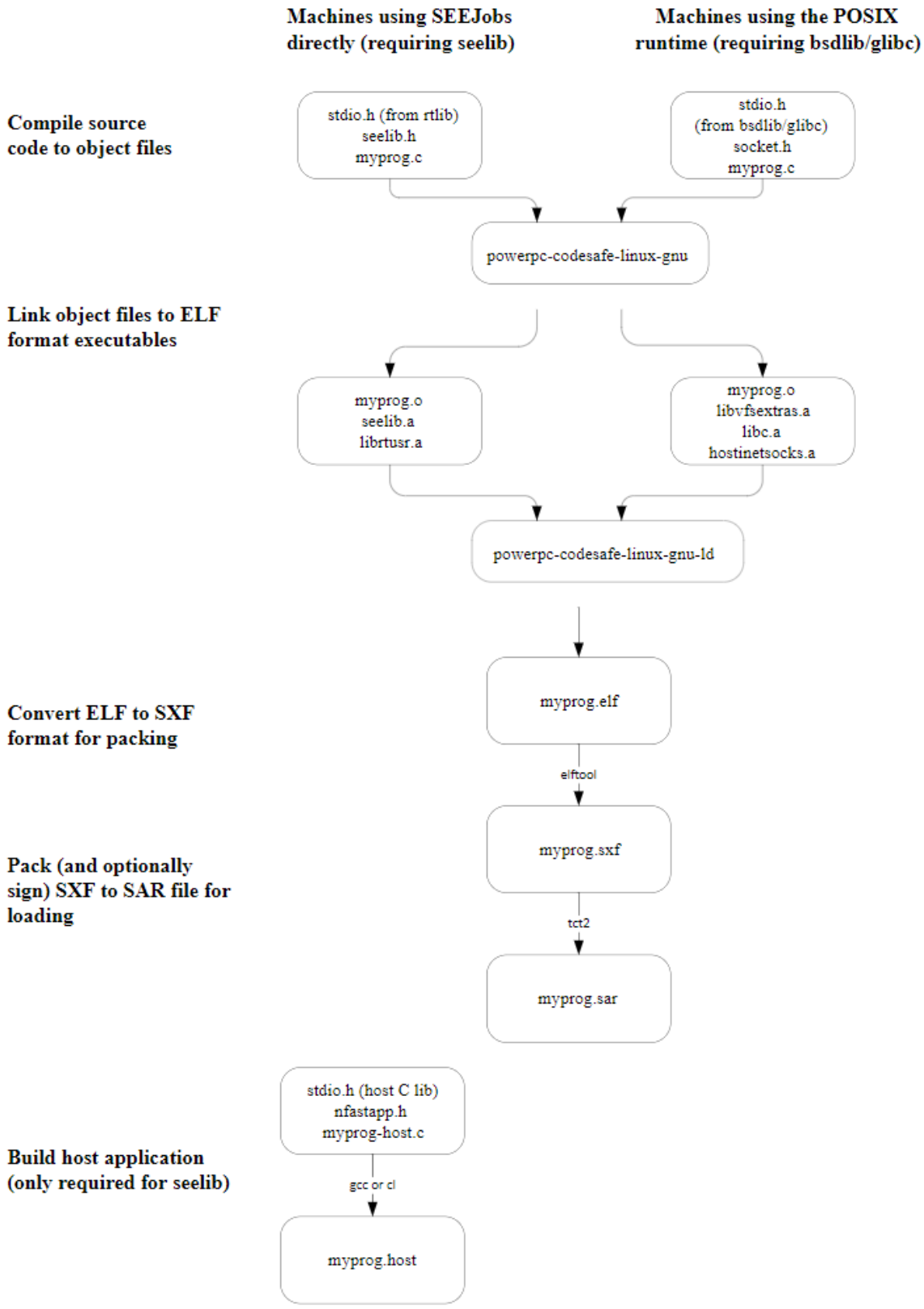
The result of the process demonstrated in this sequence of steps is that no SEE machine can use the key K_{crypto} unless at least one of the following conditions is met:

- It has been signed by the correct $K_{seemach}$ and is used in conjunction with the correct `userdata` file
- You make use of the key recovery feature.

3.1.5. Building your SEE machine and host-side application

The following steps provide an overview of the process you follow to use your application with SEE:

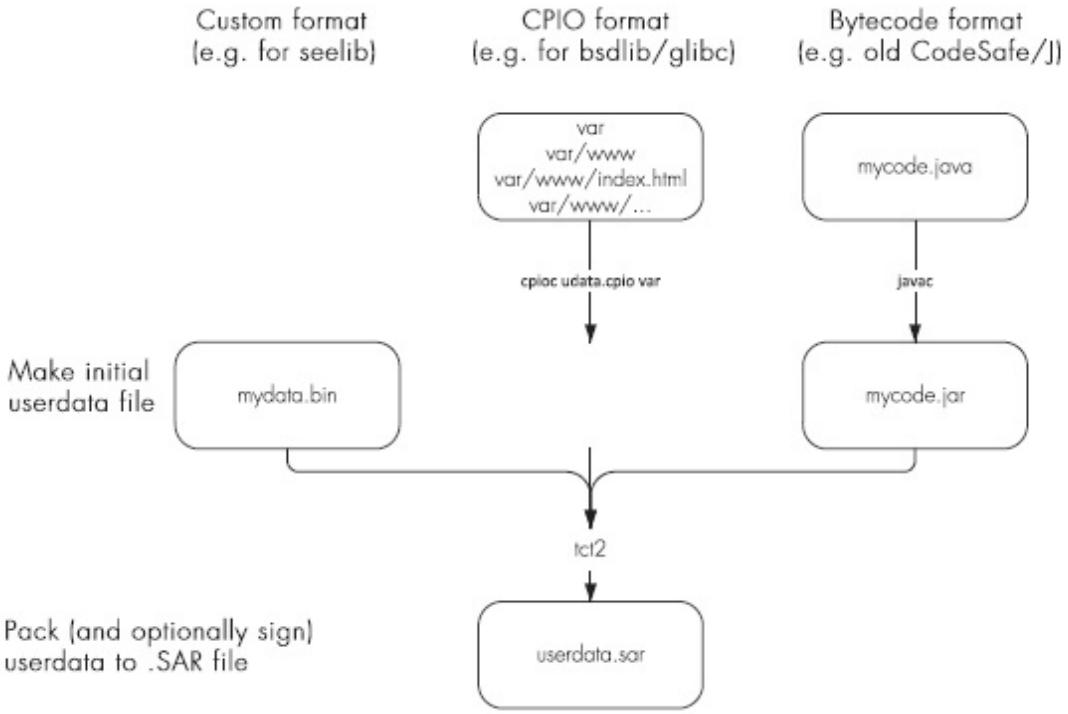
1. If you want to sign or encrypt your application, generate code-signing and confidentiality keys as applicable.
2. Compile and link the host application's source files using the native compiler on the host. See the diagram in the following step.
3. Compile and link the SEE machine source using the GCC cross compiler. See the following diagram.



4. If required, use the Trusted Code Tool (`tct2`) to sign the SEE machine with the code-signing keys. See Utilities for additional information.
5. Use the Trusted Code Tool (`tct2`) to pack the HSM files and create a SAR file. You must pack the binary file even if signatures are not required. See Utilities for additional

information.

6. Use the Trusted Code Tool (`tct2`) to pack (and, if required, sign with the code-signing keys) the `userdata` file and create a SAR file. You must pack the `userdata` file even if sig natures are not required (unless you use one of the `see-*-serv` host utilities with the `--userdata-raw` option. See Utilities for additional information.



7. If required, use the Trusted Code Tool (`tct2`) to encrypt the `userdata` file, using the con fidentiality key.
8. Place the `userdata` SAR file and the host application in an appropriate location to be used at runtime.
9. For SEE machines using the `SEElib` architecture, `userdata` file can either be either loaded automatically or can be loaded by running the `loadmache` command-line utility.

   For SEE machines that require support from a host-side `see-*-serv` utility, the host util ity loads the `userdata` file automatically.

   The following diagram shows these different methods for loading a SEE machine.

For more information, see Automatically loading a SEE machine.

4. Example SEE machines

This chapter documents the example SEE machines.

The supplied C examples consist of the source files and associated makefiles (**Linux**) and Cmake files (**Windows**) needed to compile and run the examples. To run the compiled examples correctly, you must have the latest version of the Security World for nShield. If you are using a Linux operating system, you must have version 2.22.34 or later of the HSM firmware. If you are using a Windows operating system, you must be on the latest version of the HSM firmware.

The latest versions of both the Security World for nShield and HSM firmware are supplied on the installation media.

Encrypted SEE machines are not currently supported for use with nShield Connects. When the SEEMachine binary is installed on the Connect itself for automated loading at boot, the SEE Confidentiality key is not available. However, when a client host loads a SEEMachine, it has access to the SEE Confidentiality key and can cause the binary to be decrypted. In this scenario, the Connect works fine with encrypted SEEMachine binaries.

4.1. Configure the Windows Build Environment

The Windows build environment requires that the following tools be already installed:

- CMake for Windows, minimum version 3.9.
- Visual Studio 2022 Build Tools.
- Ninja build system for Windows.

Each example is supplied with CMake files for each HSM architecture and Windows host environment.

The specifics of building the C code for the HSM architectures and the Windows host environment are described in the next sections.

In order to build the examples, you must use the Visual Studio Developer Command Prompt. This command prompt needs to be initialized to use the 64-bit compilation tools in the following manner:

1. Open a Windows command prompt, using **Run as Administrator**.
2. Navigate to the Visual Studio Build Tools installation directory. The default location for

this is `C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\VC\Auxiliary\Build`.

3. At the command prompt, execute the initialization batch file, `vcvars64.bat`. The batch file sets the required environment variables for using the 64-bit compilation tools.
4. At this point you may wish to enter the PowerShell environment. This can be done by executing the `powershell` command at the command prompt.

If you exit (close) the initialized Windows command prompt (or PowerShell), then these initialization steps must be repeated when you open a new command prompt in order to build the examples.

For information on the different library paths necessary to perform a 64-bit build of your own code, see the *nCore API Documentation* (supplied as HTML).

We strongly recommend that you familiarize yourself with the process of building the example programs supplied on the CodeSafe installation media before you attempt to adapt the makefiles to any other environments.

4.2. Examples for glibc library

This section is relevant when using a `glibc` based SEE machine with an **nShield Solo XC** or an **nShield Connect XC**.

In default CodeSafe installations, the following C examples are supplied in directories under the path `/opt/nfast/c/csd/examples/` (**Linux**) or `%NFAST_HOME%\c\csd\examples\` (**Windows**):

Location	Description
`glibsee/helloworld.c`	This example is a simple, introductory test program.
`glibsee/see-random.c`	This example demonstrates basic usage of the generic stub within SEE.
`glibsee/see-enquiry.c`	This example demonstrates host code running within SEE with no large modifications.
`glibsee/tcp-proxy.c`	This example is a multithreaded TCP-TCP proxy that forwards all connections on port 8080 to 127.0.0.1:80.

If the nShield Connect is configured to use `see-sock-serv` directly, any supplied `glibc` examples that use `see-sock-serv` can be run directly on the nShield Connect, rather than via a client machine.

The examples here show how to run a SEE machine from a command line. Alternatively, if you wish to run a SEE machine directly, please see Deploying SEE Machines.

If you are running `see-sock-serv` directly on an nShield Connect, port numbers in the examples should be modified to bind to ports within the range 8000-8999.

All supplied examples for `glibc`, both standard and SSL-related, require one of the `see-*-serv` host-side utilities. For more information about these utilities, see see-*-serve utilities.

The SEE machine type must be specified as `--machine-type=PowerPCELF` when running the tct2 tool.

4.2.1. Building the HSM-side code

1. Create a directory in your home (**Linux**) or `Documents` (**Windows**) location to contain the platform examples. For example, to create and enter a directory called `buildGLIB-mod`:

   Linux

   ```
   cd ~
   mkdir buildGLIBmod
   cd buildGLIBmod
   ```

   Windows

   ```
   cd Documents
   mkdir buildGLIBmod
   cd buildGLIBmod
   ```

2. Configure the module examples build using the command:

   Linux

   ```
   cmake -DCMAKE_TOOLCHAIN_FILE=<path to GLIB tool chain> <path to GLIB SEE examples>
   ```

   For example, using the default locations for the tool chain and the GLIB SEE examples:

   ```
   cmake -DCMAKE_TOOLCHAIN_FILE=/opt/nfast/c/csd/cmake/codesafe-linux-xc-glibsee.cmake
   /opt/nfast/c/csd/examples/
   ```

   Windows

```
cmake -G "Ninja" -DCMAKE_TOOLCHAIN_FILE=<path to GLIB tool chain> <path to GLIB SEE examples>
```

For example, using the default locations for the tool chain and the GLIB SEE examples:

```
cmake -G "Ninja" -DCMAKE_TOOLCHAIN_FILE="C:\Program Files\nCipher\nfast\c\csd\cmake\codesafe-linux-xc-glibsee.cmake" "C:\Program Files\nCipher\nfast\c\csd\examples"
```

3. Build the module examples using the command:

Linux

```
cmake --build <build output location>
```

For example:

```
cmake --build .
```

Here, the `.` specifies the location where the build products should be placed, in this case to the current directory.

Windows

```
Ninja
```

This results in the creation of a directory, `glibsee`, which contains all the compiled examples. The build process will create a file for each example, with an `.elf` suffix.

4.2.2. Helloworld example

This example source code is a simple example of an SEE machine written in C. It is *not* intended to be the basis for any real world applications. It is intended only to demonstrate how to write SEE machines in C and the use of an appropriate host utility to handle output to `stdout` and `stderr`.

4.2.2.1. Packing the SEE machine

Use the `tct2` command-line utility to convert the ELF (Executable and Linkable Format) file into a SAR (Secure or SEE ARchive) file as follows:

```
tct2 --pack --machine-type=PowerPCELF --infile=helloworld.elf --outfile=helloworld.sar
```

For additional security, you can also choose to set options in this command that sign or encrypt the file. For more information, see tct2.

4.2.2.2. Creating a userdata file

All SEE machines built with the `glibc` C library must be provided with a valid ASCII-format CPIO archive. This archive forms the base of the file system available to your SEE machine. You can use the `cpioc` command-utility that we provide to create CPIO archives of the correct type.

Although the `helloworld` example does not use its file system, you must still create and supply it with dummy `userdata` as a place-holder.

Create a dummy `userdata` file as follows:

```
echo dummy > dummy
cpioc userdata.cpio dummy
```

Output:

```
F dummy
Written 'userdata.cpio': 1 files, 0 directories, 0 errors
```

4.2.2.3. Running the example

To run the `helloworld` example on a PowerPC-based SEE machine, use the following commands:

```
see-stdoe-serv --machine helloworld.sar --userdata-raw userdata.cpio
```

Output:

```
nC SEE glibc entering main
Hello world!
```

If you are using a nShield Connect, you must also set the `--no-feature-check` option when running the `see-stdoe-serv` utility.

Before rerunning this example, run the following command to clear all HSMs:

```
nopclearfail --clear --all
```

4.2.3. SEE-Random example

This example shows basic usage of the generic stub from within SEE. It requests 128 bytes of random material from the HSM and prints the result in hexadecimal.

Before running or rerunning this example, run the following command to clear all HSMs:

```
nopclearfail --clear --all
```

The following assumes that the user is working in the directory that contains the compiled examples (both SXF and ELF files).

4.2.3.1. Packing the SEE machine

Use the `tct2` command-line utility to convert the ELF (Executable and Linkable Format) file into a SAR (Secure or SEE ARchive) file:

```
tct2 --pack --machine-type=PowerPCELF --infile=see-random.elf --outfile=see-random.sar
```

For additional security, you can also set options in this command to sign or encrypt the file. For more information, see tct2.

4.2.3.2. Running the example

To run the `SEE-Random` example on a PowerPC-based SEE machine, use the following commands:

```
see-stdoe-serv --machine see-random.sar --userdata-raw userdata.cpio
```

Output:

```
nC SEE glibc entering main
52 D1 C4 73 28 49 79 62 CD E6 64 14 1C 3B E1 B2 70 3D 6B D5 DF DE CE 7F 47 50 70 06 B6
C0 52 7F 19 3A 0A 7D E4 73 83 D8 EB F4 E5 82 F3 53 38 45 2A E3 08 49 1A 58 77 35 5F 5C
7C D9 7B 57 4A A9 C4 F4 67 C7 30 91 4A CA 0C 15 1F A7 F2 E1 2B 61 E2 3A CE EF BD FF ED
49 07 68 7B 76 D2 AC 8B 98 AA 02 FD 30 01 68 60 49 4C 0F 7E 23 7F AC EC B5 6A DE 0B CD
45 72 89 96 DD E2 96 C2 B8 7B 97 AA
```

If you are using an nShield Connect, you must also set the `--no-feature-check` option when running the `see-stdoe-serv` utility.

4.2.4. SEE-Enquiry example

This example shows how to cross-compile example code, originally written for use from the host environment, to be run within the SEE without any substantial modifications.

Before running or rerunning this example, run the following command to clear all HSMs:

```
nopclearfail --clear --all
```

The following assumes that the user is working in the directory that contains the compiled examples (both SXF and ELF files).

This example code is based on `enquiry.c` provided elsewhere in the soft ware distribution.

4.2.4.1. Packing the SEE machine

Use the `tct2` command-line utility to convert the ELF (Executable and Linkable Format) file into a SAR (Secure or SEE ARchive file):

```
tct2 --pack --machine-type=PowerPCELF --infile=see-enquiry.elf --outfile=see-enquiry.sar
```

For additional security, you can also set options in this command to sign or encrypt the file. For more information, see tct2.

4.2.4.2. Running the example

To run the `SEE-Enquiry` example on a PowerPC-based SEE machine, use the following commands:

```
see-stdoe-serv --machine see-enquiry.sar --userdata-raw userdata.cpio
```

Output:

```
nC SEE glibc entering main
Server:
 enquiry reply flags  none
 enquiry reply level  Six
 serial number        1BD7-DE7B-A370
 mode                 operational
 version              2.38.7
 speed index          4240
 rec.queue            35..152
 [etc]
```

If you are using an nShield Connect, you must also set the `--no-feature-check` option when running the `see-stdoe-serv` utility.

4.2.5. TCP proxy example

The TCP proxy example demonstrates how to set up a conduit between the local host and a destination IP address.

Before running or rerunning this example, run the following command to clear all HSMs:

```
nopclearfail --clear --all
```

The following assumes that the user is working in the directory that contains the compiled examples (both SXF and ELF files).

The default destination address is declared in the source code file `tcp-proxy.c` as follows:

```
#define BACKEND_ADDR  "127.0.0.1"
```

For the TCP proxy example to work correctly, you must change this default destination address. You can replace the default address with the IP address of any valid website.

By default, the example TCP proxy code sets the front end port to 8080 and the back end port to 80. The remainder of this example assumes the use of these values, but you can change them as necessary.

4.2.5.1. Re-building the HSM-side code

If the file `tcp-proxy.c` has been modified as described in section TCP proxy example, then the example needs to be rebuilt in order for the changes to be effective. The example can be rebuilt by executing the cmake build command from within the appropriate directory as described in section Examples for glibc library for example:

Linux

```
cd ~/buildGLIBmod
cmake --build .
```

In this example the directory path would be `~/buildGLIBmod/glibsee`.

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildGLIBMod
```

```
Ninja
```

In this example the directory path would be `C:\Users\<USER-NAME>\Documents\buildGLIBMod\glibsee\`.

If the example has been rebuilt, before continuing ensure that you are working in the directory that contains the compiled examples.

4.2.5.2. Packing the SEE machine

Use the `tct2` command-line utility to convert the ELF (Executable and Linkable Format) file into a SAR (Secure or SEE ARchive) file:

```
tct2 --pack --machine-type=PowerPCELF --infile=tcp-proxy.elf --outfile=tcp-proxy.sar
```

You can also set options in this command to sign or encrypt the file. For more information, see tct2.

4.2.5.3. Running the example

Run the example on a PowerPC-based SEE machine as follows:

```
see-sock-serv --trace --machine tcp-proxy.sar --userdata-raw userdata.cpio
```

You can check that the example is working correctly by entering the URL `http://localhost:8080/` into any browser. If the example is working correctly, the browser displays the website at the address specified in the `tcp-proxy.c` file.

4.3. Examples for SEElib

In default CodeSafe installations, the following C examples are supplied in directories under the path `/opt/nfast/c/csd/examples/csee` (**Linux**) or `%NFAST_HOME%\c\csd\examples\csee` (**Windows**).

Location	Description
`csee/hello/`	This example source code demonstrates a simple SEE machine in C and how you can use it from a C program on the host.
`csee/a3a8/`	This example code demonstrates how to write an SEE machine in C code and how to use it from a C program on the host.

Location	Description
csee/nvram/	This example shows the simple use of NVRAM in an SEE machine written in C.
csee/rtc/	This example demonstrates the use of an SEE machine written in C that implements a very simple timestamp service.
csee/tickets/	This example provides an API demonstration showing how an SEE machine can be written in C.
csee/benchmark/	This example implements a very simple utility that uses an SEE machine written in C to time stamp requests to benchmark the speed of response to requests.

We also supply a Java version of the `HelloWorld` example. This consists of the source files for host-side applications that you can run with the example SEE machines written in C in order to understand how simple SEE machines work, see About the Java example.

The `nvram`, `rtc`, and `benchmark` C examples can extract debugging information from the SEE trace buffer in all Security Worlds. If the Security World has restricted or authorized-only access to SEE debugging, the example prompts the user for the number of Administrator Cards required to gain authorization. Therefore, to avoid unnecessary exposure of the Administrator Cards, do not try to run these examples on an HSM in a production Security World. Debugging information from the trace buffer is not available for the `A3A8` or `tickets` C examples.

4.3.1. Building Linux host examples

1. Create a directory in your home location to contain the host platform examples. For example, create a directory called `buildhost`, and enter this directory:

   ```
   mkdir ~/buildhost
   cd ~/buildhost
   ```

2. Configure the host platform examples using the command:

   ```
   cmake <path to SEElib examples>
   ```

   For example:

   ```
   cmake /opt/nfast/c/csd/examples/
   ```

   Here, the location of the examples is the default location, `/opt/nfast/c/csd/examples`.

3. Build the host platform examples using the command:

```
cmake --build <build output location>
```

For example

```
cmake --build .
```

Here, the `.` specifies the location where the build products should be placed, in this case to the current directory.

This results in the creation of a directory, `csee`, which contains a subdirectory for each of the examples. For example `~/buildhost/csee/a3a8`.

4.3.2. Building Windows host examples

1. Create a directory in your `Documents` location to contain the host examples. For example, create a directory called `host`, and enter this directory:

```
cd Documents
mkdir host
cd host
```

2. Configure the host platform examples using the command:

```
cmake -G "Ninja" -DCMAKE_C_COMPILER=cl -DCMAKE_CXX_COMPILER=cl "C:\Program
Files\nCipher\nfast\c\csd\examples"
```

3. Build the host examples using the command:

```
ninja
```

This results in the creation of a directory, `csee`, which contains a subdirectory for each of the examples.

Each example's subdirectory contains a directory, `module`, which contains the complied module code.

4.3.3. Building Solo SEE module examples

1. Create a directory in your home (**Linux**) or `Documents` (**Windows**) location to contain the module examples. For example, create a directory, `buildSoloMod`, and enter this directory.

Linux

```
cd ~
mkdir buildSoloMod
cd buildSoloMod
```

Windows

```
cd Documents
mkdir buildSoloMod
cd buildSoloMod
```

2. Configure the module examples build using the command:

Linux

```
cmake -DCMAKE_TOOLCHAIN_FILE=<path to Solo + module tool chain> <path to CSEE examples>
```

For example, using default locations for the Solo + module tool chain and the CSEE examples:

```
cmake -DCMAKE_TOOLCHAIN_FILE=/opt/nfast/c/csd/cmake/codesafe-linux-solo-csee.cmake
/opt/nfast/c/csd/examples/
```

Windows

```
cmake -G "Ninja" -DCMAKE_TOOLCHAIN_FILE="C:\Program Files\nCipher\nfast\c\csd\cmake\codesafe-linux-
solo-csee.cmake" "C:\Program Files\nCipher\nfast\c\csd\examples"
```

3. Build the modules using the command:

Linux

```
cmake --build <build output location>
```

For example:

```
cmake --build .
```

Here the `.` specifies the location where the build products should be placed, in this case to current directory.

Windows

```
Ninja
```

This will result in the creation of a directory, `csee`, which contains a subdirectory for

each of the examples. Each example's subdirectory contains a directory, `module`, which contains the compiled module code. For example, `~/buildSoloMod/csee/a3a8/module`. The compiled module executables have the suffix `sxf`.

4.3.4. Building Solo XC SEE module examples

1. Create a directory in your home (**Linux**) or `Documents` (**Windows**) location to contain the module platform examples. For example, create a directory, `buildXCmod`, and enter this directory:

   Linux

   ```
   cd ~
   mkdir buildXCmod
   cd bukdXCmod
   ```

   Windows

   ```
   cd Documents
   mkdir buildXCmod
   cd buildXCmod
   ```

2. Configure the module examples build using the command:

   Linux

   ```
   cmake -DCMAKE_TOOLCHAIN_FILE=<path to Solo XC module tool chain> <path to SEElib examples>
   ```

   For example, using the default locations for the Solo XC module tool chain and the SEELib examples, the command would be:

   ```
   cmake -DCMAKE_TOOLCHAIN_FILE=/opt/nfast/c/csd/cmake/codesafe-linux-xc-csee.cmake
   /opt/nfast/c/csd/examples/
   ```

   Windows

   ```
   cmake -G "Ninja" -DCMAKE_TOOLCHAIN_FILE="C:\Program Files\nCipher\nfast\c\csd\cmake\codesafe-linux-xc-
   csee.cmake" "C:\Program Files\nCipher\nfast\c\csd\examples"
   ```

3. Build the module examples using the command:

   Linux

   ```
   cmake --build <build output location>
   ```

   For example:

```
cmake --build .
```

Here the `.` specifies the location where the build products should be placed, in this case to current directory.

Windows

```
Ninja
```

This will result in the creation of a directory, `csee`, which contains a subdirectory for each of the examples Each example's subdirectory contains a directory, `module`, which contains the compiled module code. For example, `~/buildXCMod/csee/a3a8/module`. The compiled module executables have the suffix `.elf`.

4.3.5. Example: Hello-World

This example source code demonstrates a simple SEE machine in C and how you can use it from a C program on the host. The SEE machine examines the characters in the SEE job passed to it and replaces each lowercase alphabetic character with the corresponding uppercase character, returning the result as the SEE job reply. Additionally, if the SEE World is created with a `userdata` file, any characters found in the `userdata` file are replaced in the input SEE job with the character `X`.

The `Hello-World` example is *not* intended to be the basis for any real world applications. It is intended only to demonstrate how to write SEE machines in C and host-side use of an SEE machine by code written in C.

There is also an example of the host-side code written in Java, supplied in the `nCipherKM-SEE-Examples.jar` found in `/opt/nfast/java/examples/` (**Linux**) or `%NFAST_HOME%\java\examples\` (**Windows**).

4.3.5.1. Signing, packing, and loading the SEE machine

1. Generate a key with which to sign the SEE machine:

```
generatekey -m 1 seeinteg
```

2. Complete the prompts as follows:

```
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (RSA, DSA, ECDSA, KCDSA) [RSA] >
```

```
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
plainname: Key name? [] > hellomachine
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation      Operation to perform            generate
 application    Application                     seeinteg
 protect        Protected by                    token
 slot           Slot to read cards from         0
 recovery       Key recovery                    yes
 verify         Verify security of key          yes
 type           Key type                        RSA
 size           Key size                        2048
 pubexp         Public exponent for RSA         key (hex)
 plainname      Key name                        hellomachine
 nvram          Blob in NVRAM (needs ACS)       no

Loading `dev-ocs':
  Module 1: 0 cards of 1 read
  Module 1 slot 0: `dev-ocs' #1
  Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
Key successfully generated.
Path to key: <path-to-key>
```

Where `<path-to-key>` is `/opt/nfast/kmdata/local/key_seeinteg_hellomachine` (**Linux**) or `C:\ProgramData\nCipher\Key Management Data\local\key_seeinteg_hellomachine` (**Windows**).

3. Change to the module directory.

   For **nShield Solo**:

   Linux

   ```
   cd ~/buildSoloMod/csee/hello/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\hello\module
   ```

   For **nShield Solo XC**:

   Linux

   ```
   cd ~/buildSoloXC/csee/hello/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\hello\module
   ```

4. Use the `tct2` command line utility to convert the file into a SAR file.

For **nShield Solo**:

Convert the `hello.sxf` file to a SAR file:

Linux

```
tct2 --sign-and-pack --is-machine -i hello.sxf --machine-type=PowerPCSXF -o hello.sar -k hellomachine
```

Windows

```
tct2 -m 1 --sign-and-pack --is-machine -i .\hello.sxf --machine-type=PowerPCSXF -o hello.sar -k
hellomachine
```

Output:

```
Signing machine as `PowerPCSXF'.

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

For **nShield Solo XC**:

Convert the `hello.elf` file to a SAR file:

Linux

```
tct2 --sign-and-pack --is-machine -i hello.elf --machine-type=PowerPCELF -o hello.sar -k hellomachine
```

Windows

```
tct2 --sign-and-pack --is-machine -i .\hello.elf --machine-type=PowerPCELF -o hello.sar -k
hellomachine
```

Output:

```
Signing machine as `PowerPCELF'.

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

For more information about this command, see tct2.

5. Load the SEE machine into the HSM by running the command:

```
loadmache -m 1 hello.sar
```

This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment, you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. See Automatically loading an SEE machine.

4.3.5.2. Preparing example userdata

You do not need to create real `userdata` for this example. Instead, you can simply pack a small text file with `tct2` and pass the packed file to the SEE machine to serve as `userdata`.

However, you can also choose to create and sign a real `userdata` file in the same way as for the `A3A8` example; see A3A8 example

When you run the `Hello-World` example, because the characters in the `userdata` you supply are converted from lower case to replaced by the character `X` in the output file, including a new line sequence in the `user-data` can produce unexpected results.

4.3.5.3. Running the example

To run the C example change to the host application directory by running the command:

Linux

```
cd ~/buildHost/csee/hello/hostside
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\hostside\csee\hello\hostside
```

Pack the desired user data in the SAR file suitable for loading onto the HSM. Optionally, you could use the Trusted Code Tool (`tct2`) to create a signed and packed SAR file for this step.

4.3.5.4. Usage

The `hello` example program has the following arguments:

```
hello <FILENAME> [<USERDATA>.sar]
```

FILENAME

This parameter is the name of the input file that contains the source string.

USERDATA

This optional parameter is the name of a file that contains letters to be replaced by the ASCII character `X` in the output file.

4.3.5.4.1. What the code actually does

The host-side C code performs the following tasks:

1. It prompts the user to supply a file name and an optional *USERDATA* file.
2. It sends the string in the file, converted if necessary to standard output.

The HSM-side code awaits jobs from the host and performs the following:

1. It transforms the contents of the input file, capitalizing all input and replacing any characters that appear in the optional *USERDATA* file with an ASCII character `X`.
2. It sends the result as output.

4.3.6. A3A8 example

This example code demonstrates how to write an SEE machine in C code and how to use it from a C program on the host.

The `A3A8` example is *not* intended to be the basis for any real world applications. The algorithm used, known as ACOMP128, has been shown to be insecure and is not appropriate for production use. It is used here only to demonstrate the implementation of an algorithm in an SEE application, not to endorse it in any way.

This example does not support debugging when the SEE debug level is set to `Generate Authorization Key`.

The SEE machine is used to process data with the A3/A8 algorithm in conjunction with a Triple-DES key as follows:

1. Data comes in the form of a sequence of 16-byte input values.
2. These values are split into two 8-byte halves that are each Triple-DES ECB decrypted with the master key and reassembled to give a 16-byte key.
3. Then a 16-byte random value is generated and, along with the 16-byte key, is fed into

the A3/A8 algorithm to produce a 12-byte output value.

4. The output from the HSM consists of a sequence of 28-byte blocks comprising the ran dom value and the output value.

There is also an example of the host-side code written in Java, supplied in the `nCipherKM-SEE-Examples.jar` found in `opt/nfast/java/examples` (**Linux**) or `%NFAST_HOME%\java\examples` (**Windows**).

4.3.6.1. Signing, packing, and loading the SEE machine

To sign, pack, and load the SEE machine:

1. Generate a key with which to sign the SEE machine:

```
generatekey -m 1 seeinteg
```

2. Complete the prompts as follows:

```
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (RSA, DSA, ECDSA, KCDSA) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
plainname: Key name? [] > a3a8machine
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation    Operation to perform              generate
 application  Application                       seeinteg
 protect      Protected by                      token
 slot         Slot to read cards from           0
 recovery     Key recovery                      yes
 verify       Verify security of key            yes
 type         Key type                          RSA
 size         Key size                          2048
 pubexp       Public exponent for RSA key (hex)
 plainname    Key name                          a3a8machine
 nvram        Blob in NVRAM (needs ACS)         no

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: <path-to-key>
```

`<path-to-key>` is `/opt/nfast/kmdata/local/key_seeinteg_a3a8machine` (**Linux**) or `C:\ProgramData\nCipher\Key Management Data\local\key_seeinteg_a3a8machine` (**Windows**).

3. Change to the directory by running the command:

For **nShield Solo**

Linux

```
cd ~/buildSoloMod/csee/a3a8/module
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\a3a8\module
```

For **nShield Solo XC**

Linux

```
cd ~/buildXCMod/csee/a3a8/module
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\a3a8\module
```

4. Use the `tct2` command line utility to convert the file into a SAR file.

```
$ generatekey -m 1 seeinteg
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (RSA, DSA, ECDSA, KCDSA) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
plainname: Key name? [] > a3a8machine
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation   Operation to perform                 generate
 application  Application                          seeinteg
  protect      Protected by                         token
 slot         Slot to read cards from              0
 recovery     Key recovery                         yes
 verify       Verify security of key               yes
 type         Key type                             RSA
 size         Key size                             2048
 pubexp       Public exponent for RSA key (hex)
 plainname    Key name                             a3a8machine
 nvram Blob in NVRAM (needs ACS) no

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: /opt/nfast/kmdata/local/key_seeinteg_a3a8machine
```

5. Change to the directory by running the command:

For **nShield Solo**

Convert the `a3a8mach.sxf` file into a SAR file.

```
tct2 -m 1 --sign-and-pack --is-machine -i a3a8mach.sxf --machine-type=PowerPCSXF -o a3a8mach.sar -k
a3a8machine
```

Output:

```
Signing machine as `PowerPCSXF'.

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

For **nShield Solo XC**

Convert the `a3a8mach.elf` file into a SAR file.

```
tct2 -m 1 --sign-and-pack --is-machine -i a3a8mach.elf --machine-type=PowerPCELF -o a3a8mach.sar -k
a3a8machine
```

Output:

```
Signing machine as `PowerPCELF'.

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

6. Load the SEE machine into the HSM by running the command:

```
loadmache -m 1 a3a8mach.sar
```

This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment, you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. See Automatically loading an SEE machine

4.3.6.2. Creating and signing userdata

To create and sign the `userdata` file:

1. Change to the host-side code directory by running the command:

   Linux

   ```
   $ cd ~/buildhost/csee/a3a8/hostside
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\hostside\csee\a3a8\hostside
   ```

2. Generate a key with which to sign a dummy `userdata` file for the example by running the command:

   ```
   generatekey -m 1 seeinteg
   ```

3. Complete the prompts as follows:

   ```
   recovery: Key recovery? (yes/no) [yes] >
   type: Key type? (RSA, DSA, ECDSA, KCDSA) [RSA] >
   size: Key size? (bits, minimum 1024) [2048] >
   OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
   >
   plainname: Key name? [] > a3a8userdata
   nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
   key generation parameters:
    operation    Operation to perform              generate
    application  Application                       seeinteg
    protect      Protected by                      token
    slot         Slot to read cards from           0
    recovery     Key recovery                      yes
    verify       Verify security of key            yes
    type         Key type                          RSA
    size         Key size                          2048
    pubexp       Public exponent for RSA key (hex)
    plainname    Key name                          a3a8userdata
    nvram        Blob in NVRAM (needs ACS)         no

   Loading `dev-ocs':
    Module 1: 0 cards of 1 read
    Module 1 slot 0: `dev-ocs' #1
    Module 1 slot 0:- passphrase supplied - reading card
   Card reading complete.

   Key successfully generated.
   Path to key: <path-to-key>
   ```

   `<path-to-key>` is `/opt/nfast/kmdata/local/key_seeinteg_a3a8userdata` (**Linux**) or `C:\ProgramData\nCipher\Key Management Data\local\key_seeinteg_a3a8userdata` (**Windows**).

4. Create a dummy `userdata` file. Because the A3/A8 algorithm does not use the initialization data, the dummy `userdata` need contain only one arbitrary character to use as `user data`.

5. Use the `tct2` command-line utility to sign and pack a dummy `userdata` file for the example:

   For **nShield Solo**

   Linux

   ```
   tct2 --sign-and-pack --machine-type=PowerPCSXF --infile a3a8userdata --outfile=a3a8userdata.sar
   --machine-key-ident=a3a8machine -k a3a8userdata
   ```

   Output:

   ```
   Loading `ocs-dev':
    Module 1: 0 cards of 1 read
    Module 1 slot 0: `ocs-dev' #1
    Module 1 slot 0:- passphrase supplied - reading card
   Card reading complete.[sudo] password for XXX:
   ```

   Windows

   ```
   tct2 --sign-and-pack --machine-type=PowerPCSXF --infile=a3a8userdata --outfile=a3a8userdata.sar
   --machine-keyident=a3a8machine -k a3a8userdata
   ```

   Output:

   ```
   No module specified, using 1
   Signing machine as 'PowerPCSXF'.
   ```

   For **nShield Solo XC**:

   Linux

   ```
   tct2 --sign-and-pack  --machine-type=PowerPCELF --infile a3a8userdata --outfile a3a8userdata.sar
   --machine-key-ident=a3a8machine -k a3a8userdata
   ```

   Output:

   ```
   Loading `ocs-dev':
    Module 1: 0 cards of 1 read
    Module 1 slot 0: `ocs-dev' #1
    Module 1 slot 0:- passphrase supplied - reading card
   Card reading complete.
   ```

   Windows

   ```
   tct2 --sign-and-pack --machine-type=PowerPCELF --infile=a3a8userdata --outfile=a3a8userdata.sar
   --machine-keyident=a3a8machine -k a3a8userdata
   ```

   Output:

```
No module specified, using 1
Signing machine as 'PowerPCELF'.
```

For more information about this command, see tct2

4.3.6.2.1. Running and testing the example

The a3test example application takes the following arguments:

```
a3test [-m <MODULEID>] <USERDATA>.sar
```

-m <MODULEID>

This option specifies the ModuleID of the HSM to use.

<USERDATA>.sar

This parameter specifies a userdata file (packed as a SAR) to use.

Thus, you can run the a3test program created in this example with a command of the form:

Linux

```
./a3test -m 1 a3a8userdata.sar
```

Windows

```
a3test -m 1 a3a8userdata.sar
```

The a3test example then processes data for approximately 20 seconds. If the example program runs successfully, its final output is of the form:

```
Getting Sarfile info (400 bytes)....
Creating world: init status was 0 (OK)
Making Master Key:
Get ticket.......
Sending ticket to SEEWorld:
181000 triples, 21 sec
Releasing context
Thank you for watching. The end.
```

If the output from a3test takes any other form, this indicates an error. In case of an error, use the enquiry command-line utility to check:

- Whether the correct firmware is installed
- Whether the correct server is running
- Whether the HSM is in the operational state.

4.3.6.3. NVRAM example

The NVRAM example shows the simple use of NVRAM in an SEE machine written in C. It uses a file in NVRAM as a sort of postage meter. The contents of the file are interpreted as a little-endian integer that determines how many 'stamps' can be issued. Each time the host program is invoked, it requests one or more stamps from the machine, and the NVRAM counter is decreased accordingly.

The NVRAM example is *not* intended to be the basis for any real world applications. It is intended only to demonstrate how to write SEE machines in C that access the HSM's NVRAM.

4.3.6.3.1. Signing, packing, and loading the SEE machine

To sign, pack, and load the SEE machine:

1. Generate a key with which to sign the SEE machine:

   ```
   generatekey seeinteg
   ```

2. Complete the prompts as follows:

   ```
   module: Module to use? (1, 2) [1] >
   recovery: Key recovery? (yes/no) [yes] >
   type: Key type? (RSA, DSA) [RSA] >
   size: Key size? (bits, minimum 1024) [1024] >
   OPTIONAL: pubexp: Public exponent for RSA key (in hex)? []
   >
   plainname: Key name? [] > nvrammachine
   key generation parameters:
    operation    Operation to perform                 generate
    application  Application                          seeinteg
    module       Module to use                        1
    protect      Protected by                         token
    slot         Slot to read cards from              0
    recovery     Key recovery                         yes
    verify       Verify security of key               yes
    type         Key type                             RSA
    size         Key size                             1024
    pubexp       Public exponent for RSA key (in hex)
    plainname    Key name                             nvrammachine
   Key successfully generated.
   Path to key: <path-to-key>
   ```

   <path-to-key> is /opt/nfast/kmdata/local/key_seeinteg_nvrammachine (**Linux**) or %NFAST_KMDATA%\local\key_seeinteg_nvrammachine (**Windows**).

3. Change to the directory by running the command:

   For **nShield Solo**

Linux

```
cd ~/buildSoloMod/csee/nvram/module
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\nvram\module
```

For **nShield Solo XC**

Linux

```
cd ~/buildXCMod/csee/nvram/module
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\nvram\module
```

4. Use the `tct2` command line utility to convert the file.

For **nShield Solo**

Convert the `nvram.sxf` file into a SAR file.

Linux

```
tct2 --sign-and-pack --is-machine -i nvram.sxf --machine-type=PowerPCSXF -o nvram.sar -k nvrammachine
```

Windows

```
tct2 -m 1 --sign-and-pack --is-machine -i nvram.sxf --machine-type=PowerPCSXF -o nvram.sar -k
nvrammachine
```

Output:

```
Signing machine as 'PowerPCSXF'.

Loading `ocs-dev':
Module 1: 0 cards of 1 read
Module 1 slot 0: `ocs-dev' #1
Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

For **nShield Solo XC**

Convert the `nvram.elf` file into a SAR file.

Linux

```
tct2 --sign-and-pack  --is-machine -i nvram.elf --machine-type=PowerPCELF -o nvram.sar -k nvrammachine
```

Windows

```
tct2 -m 1 --sign-and-pack --is-machine -i nvram.elf --machine-type=PowerPCELF -o nvram.sar -k
nvrammachine
```

Output:

```
Signing machine as 'PowerPCELF'.

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
```

For more information about this command, see tct2.

5. Load the packed SEE machine into the HSM by running the command:

```
loadmache nvram.sar
```

This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment, you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. For information about configuration files, see HSM and client configuration files (network-attached HSMs) or Hardserver configuration files (PCIe and USB HSMs).

4.3.6.3.2. Creating NVRAM and userdata files

You must now use the `setup` example application to create:

- An NVRAM file
- A `userdata` file that contains only the exact name of the specified NVRAM file.

1. Change to the host-side application directory by running the command:

Linux

```
cd ~/buildhost/csee/nvram/hostside
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\hostside\csee\nvram\host
```

2. Create these files by running the setup command with 'root' (**Linux**) or Administrator (**Windows**) privileges:

   Linux

   ```
   ./setup nvramfile 100 nvramuserdata
   ```

   Windows

   ```
   setup.exe nvramfile 100 nvramuserdata
   ```

3. Complete the on-screen instructions:

   ```
   Please insert the next administrator card and press enter.
   Please enter card passphrase:
   allocated NVRAM file `nvramfile'.
   ```

4.3.6.3.3. setup

The `setup` example application takes the following arguments:

```
setup [-k|--key <APPNAME>,<IDENT>] <nvram-filename> <stamp-count> <userdatafile>
```

`-k|--key <APPNAME>,<IDENT>`

This option specifies a signing key identified by *APPNAME* and *IDENT*. Specifying a signing key creates an NVRAM file that can only be accessed with authorization from that key (for example, by signing the `userdata` with the same key). A signing key is optional.

`<nvram-filename>`

This parameter specifies the name of an NVRAM file to create. The name must contain no more than 11 characters.

`<tampcount>`

This parameter specifies the number of stamps to issue.

`<userdatafile>`

This parameter specifies the name of the `userdata` SAR file created when the `setup` example application is run.

You can also use the `setup` example application to delete an existing NVRAM file. To delete a file, run `setup` with the `--delete` option, as follows:

```
setup --delete <nvram-filename>
```

In this case, `setup` deletes the NVRAM file specified by *nvram-filename*.

4.3.6.3.4. Signing and packing the userdata

Run the Trusted Code Tool (`tct2`) to sign and pack the created `userdata` file you created with the `setup` example application:

Linux

```
tct2 -m 1 --pack --infile nvramuserdata --outfile nvramuserdata.sar
```

Windows

```
tct2 --pack --infile nvramuserdata --outfile nvramuserdata.sar
```

If the NVRAM file created by the `setup` example application is bound to a key (that is, if you specified the `-k|--key` option when running `setup`), use that same key when signing the `userdata` file with `tct2`.

4.3.6.3.5. Running and testing the example

Run the `nvram` example application as follows:

Linux

```
./nvram ./nvramuserdata.sar 50
```

Windows

```
nvram.exe nvramuserdata.sar 50
```

Output:

```
SEEJob: read 1 bytes...
Stamp Request Accepted.
SEEJob: read 1 bytes...
Stamp Request Accepted.
SEEJob: read 1 bytes...
.
.
.
```

4.3.6.3.6. nvram

The `nvram` example application takes the following arguments:

```
nvram <userdatafile>.sar [<iterations>]
```

`<userdatafile>.sar`

This parameter specifies the name of the `userdata` SAR file to use. Normally, this file has been created by the `setup` example application (its name specified by that utility's *user-datafile* parameter).

`<iterations>`

This parameter specifies an integer that is the amount by which the `nvram` example application is to decrease its counter (as it issues virtual stamps).

4.3.6.3.7. What the code actually does

The host-side code performs the following tasks in order:

1. It allocates an NVRAM file with an access control list that requires the permission of a specified key for reading or writing.
2. It requests the name of a file to be loaded as a packed user data block and, optionally, the number of virtual stamps to request.

The HSM-side code awaits jobs from the host and returns a single byte to indicate whether or not a stamp has been issued.

4.3.7. Example: RTC

This source code provides an example of an SEE machine written in C that implements a very simple timestamp service.

Your SEE-Ready HSM must have an onboard real-time clock for this example to run correctly, and you must have set the clock using the `rtc` command-line utility.

The `rtc` example code is deficient in a number of ways and is *not* intended to be the basis for any real world applications. It is intended only to demonstrate some important concepts in writing SEE machines in C to perform time-stamping.

4.3.7.1. Signing, packing, and loading the SEE machine

To sign, pack, and load the SEE machine:

1. Generate a key with which to sign the SEE machine:

```
generatekey -m 1 seeinteg
```

2. Complete the prompts as follows:

```
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (RSA, DSA, ECDSA, KCDSA) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
plainname: Key name? [] > rtccode
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation    Operation to perform              generate
 application  Application                        seeinteg
 protect      Protected by                       token
 slot         Slot to read cards from            0
 recovery     Key recovery                       yes
 verify       Verify security of key             yes
 type         Key type                           RSA
 size         Key size                           2048
 pubexp       Public exponent for RSA key (hex)
 plainname    Key name                           rtccode
 nvram        Blob in NVRAM (needs ACS)          no

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.
Key successfully generated.
Path to key: <path-to-key>
```

`<path-to-key` is `/opt/nfast/kmdata/local/key_seeinteg_rtccode` (**Linux**) or `C:\ProgramData\nCipher\Key Management Data\local\key_seeinteg_rtccode` (**Windows**).

3. Change to the directory by running the command:

   For **nShield Solo**:

   Linux

   ```
   cd ~/buildSoloMod/csee/rtc/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\rtc\module
   ```

   For **nShield Solo XC**:

Linux

```
cd ~/build-XC/csee/rtc/module
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\rtc\module
```

4. Use the `tct2` command-line utility to convert the file into a SAR file.

   For **nShield Solo**

   ```
   tct2 -m 1 --sign-and-pack --is-machine -i rtc.sxf --machine-type=PowerPCSXF -o rtc.sar -k rtccode
   ```

   Output:

   ```
   Signing machine as `PowerPCSXF'.

   Loading `ocs-dev':
    Module 1: 0 cards of 1 read
    Module 1 slot 0: `ocs-dev' #1
    Module 1 slot 0:- passphrase supplied - reading card


   Card reading complete.
   ```

   For **nShield Solo XC**:

   ```
   tct2 -m 1 --sign-and-pack --is-machine -i rtc.elf --machine-type=PowerPCELF -o rtc.sar -k rtccode
   ```

   Output:

   ```
   Signing machine as `PowerPCELF'.

   Loading `ocs-dev':
    Module 1: 0 cards of 1 read
    Module 1 slot 0: `ocs-dev' #1
    Module 1 slot 0:- passphrase supplied - reading card


   Card reading complete.
   ```

5. Load the packed SEE machine into the HSM by running the command:

   ```
   loadmache rtc.sar
   ```

This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment,

> you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. For information about configuration files, see HSM and client configuration files (network-attached HSMs) or Hard-server configuration files (PCIe and USB HSMs).

4.3.7.1.1. rtc

The `rtc` example application takes the following arguments:

```
rtc [-y|--verify <file>] [-a|--userdata <SEEDATA>] <userdatafile> <APPNAME>,<IDENT>
```

`-y|---verify`

This option verifies the returned time-stamp for the file named *file*.

`-a|---userdata <SEEDATA>`

This option specifies use of the file *SEEDATA* for SEE `userdata`.

`<userdatafile>`

This parameter specifies a `userdata` file that contains at least one character.

`<APPNAME>,<IDENT>`

These parameters specify the *APPNAME* and *IDENT* of the key for the `rtc` example application to use.

4.3.7.1.2. Running the example

1. Enter the host-side application directory by running the command:

   Linux

   ```
   cd ~/buildhost/csee/rtc/hostside/
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\host\csee\rtc\hostside
   ```

2. Create the test userdata file to be time-stamped by running the command:

   Linux

   ```
   cp /opt/nfast/c/csd/examples/csee/rtc/host/rtc.c ./mytestuserdata
   ```

Windows

```
copy "C:\Program Files\nCipher\nfast\c\csd\examples\csee\rtc\hostside\rtc.c" mytestuserdata
```

3. Generate an RSA key for the RTC example to use by running the command and completing the prompts in the output as follows:

Linux

```
generatekey simple
```

Output:

```
protect: Protected by? (token, module) [token] >
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,HMACRIPEMD160, HMACSHA1,
HMACSHA256, HMACSHA384, HMACSHA512, HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
ident: Key identifier? [] > rtctest
plainname: Key name? [] > rtctest
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, nvrammachine, a3a8userdata, hellomachine, rtccode) []
>
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
operation     Operation to perform              generate
application   Application                       simple$ ./rtc mytestuserdata simple,rtctest >
mytestuserdata.stamp
Please insert the next operator card and press enter.
Please enter card passphrase:
rtc: timestamp issued.

protect       Protected by                      token
slot          Slot to read cards from           0
recovery      Key recovery                      yes
verify        Verify security of key            yes
type          Key type                          RSA
size          Key size                          2048
pubexp        Public exponent for RSA key (hex)
ident         Key identifier                    rtctest
plainname     Key name                          rtctest
seeintegname  SEE integrity key
nvram         Blob in NVRAM (needs ACS)         no

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: /opt/nfast/kmdata/local/key_simple_rtctest
```

Windows

```
generatekey -m 1 simple
```

Output:

```
protect: Protected by? (token, module) [token] >
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,
                 HMACRIPEMD160, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512,
                 HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
ident: Key identifier? [] > rtctest
plainname: Key name? [] > rtctest
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, a3a8userdata, hellomachine, rtccode) [] >
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation     Operation to perform              generate
 application   Application                       simple
 protect       Protected by                      token
 slot          Slot to read cards from           0
 recovery      Key recovery                      yes
 verify        Verify security of key            yes
 type          Key type                          RSA
 size          Key size                          2048
 pubexp        Public exponent for RSA key (hex)
 ident         Key identifier                    rtctest
 plainname     Key name                          rtctest
 seeintegname  SEE integrity key
 nvram         Blob in NVRAM (needs ACS)         no

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: C:\ProgramData\nCipher\Key Management Data\local\key_simple_rtctest
```

The `rtc` example application only supports the use of RSA keys.

4. Run the RTC example by executing the following command:

Linux

```
./rtc mytestuserdata simple,rtctest > mytestuserdata.stamp
```

Windows

```
rtc.exe mytestuserdata simple,rtctest > mytestuserdata.stamp
```

5. Complete the on-screen instructions:

```
Please insert the next operator card annvramd press enter.
Please enter card passphrase:
rtc: timestamp issued.
```

4.3.7.1.3. What the code actually does

The host-side code performs the following tasks in order:

1. It sends a session key to the HSM.
2. When a time-stamped command is returned, it verifies the time-stamp using the session key.

The HSM-side code performs the following tasks in order:

1. It awaits a job from the host.
2. It time-stamps the contents of the job and signs the result with the session key.
3. It returns the job to the host.

4.3.8. Example: Tickets

This example source code is an API demonstration showing how an SEE machine can be written in C.

The `Tickets` example is *not* intended to be the basis for any real world applications. In particular, it does not support the loading of keys protected by card sets with the `-k` option. It is intended to demonstrate:

- How to write SEE machines in C
- Simple, custom-built marshalling and unmarshalling of jobs
- The use of tickets. See Internals for information about key tickets; also, for information about the consumption of single ticket, see Loading stored keys.

Windows only

`%NFAST_HOME%\c\csd\examples\csee\tickets\`

The C code consists of the following parts:

- In the `module` directory, the source for the SEE machine:
  - `armtickets.c` (SEE machine start-up and job-processing threads)
  - `usrjobs.c` (job processing code)
- In the `host` directory, source for the host application:
  - `hosttickets.c` (starts the SEE machine, sends jobs and traces debug).
- The `common` directory contains a simple header file (`common.h`) for shared data structures between the HSM and the host code.

Sample makefiles are provided for building the HSM and host-side code (`Makefile-host`)

and can be found in their respective directories.

4.3.8.1. Signing, packing, and loading the SEE machine

To sign, pack, and load the SEE machine:

1. Change to the module directory by running the command:

   For **nShield Solo**

   Linux

   ```
   cd ~/buildSoloMod/csee/tickets/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\tickets\module
   ```

   For **nShield Solo XC**

   Linux

   ```
   cd ~/buildXCMod/csee/tickets/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\tickets\module
   ```

2. Use the `tct2` command line utility to convert the file into a SAR file.

   For **nShield Solo**

   Convert the `armtickets.sxf` file into a SAR file.

   ```
   tct2 -m 1 --pack --infile armtickets.sxf --outfile armtickets.sar
   ```

   For **nShield Solo XC**

   Convert the `armtickets.elf` file into a SAR file.

   ```
   tct2 -m 1 --pack --infile armtickets.elf --outfile armtickets.sar
   ```

3. Load the SEE machine into the HSM by running the command:

   ```
   loadmache armtickets.sar
   ```

This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment, you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. For information about configuration files, see HSM and client configuration files (network-attached HSMs) or Hard-server configuration files (PCIe and USB HSMs).

4. Generate a key for the example to use by running the command and completing the prompts in the output as follows:

Linux

```
generatekey simple
```

Output:

```
protect: Protected by? (token, module) [token] >
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,
HMACRIPEMD160, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512,
HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
ident: Key identifier? [] > ticketkey
plainname: Key name? [] > ticketkey
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, nvrammachine, a3a8userdata, hellomachine, rtccode) []
>
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
operation     Operation to perform              generate
application   Application                       simple
protect       Protected by                      token
slot          Slot to read cards from           0
recovery      Key recovery                      yes
verify        Verify security of key            yes
type          Key type                          RSA
size          Key size                          2048
pubexp        Public exponent for RSA key (hex)
ident         Key identifier                    ticketkey
plainname     Key name                          ticketkey
seeintegname  SEE integrity key
nvram         Blob in NVRAM (needs ACS)         no

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: /opt/nfast/kmdata/local/key_simple_ticketkey
```

Windows

```
generatekey -m 1 simple
```

Output:

```
protect: Protected by? (token, module) [token] >
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,
                 HMACRIPEMD160, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512,
                 HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
ident: Key identifier? [] > ticketkey
plainname: Key name? [] > ticketkey
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, a3a8userdata, hellomachine, rtccode) [] >
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation     Operation to perform              generate
 application   Application                       simple
 protect       Protected by                      token
 slot          Slot to read cards from           0
 recovery      Key recovery                      yes
 verify        Verify security of key            yes
 type          Key type                          RSA
 size          Key size                          2048
 pubexp        Public exponent for RSA key (hex)
 ident         Key identifier                    ticketkey
 plainname     Key name                          ticketkey
 seeintegname  SEE integrity key
 nvram         Blob in NVRAM (needs ACS)         no

Loading `dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: C:\ProgramData\nCipher\Key Management Data\local\key_simple_ticketkey
```

4.3.8.2. hosttickets

The `hosttickets` example application takes the following arguments:

```
hosttickets [-f|--file <userdatafile>][-k|--key <APPNAME>,<IDENT>]
```

`-k|--key <APPNAME>,<IDENT>`

These options specify a Security World key.

For the public/private key pair, a Security World key can be specified with the `-k` option. The specified Security World key must be an RSA key of the type `simple` that is not tied to an SEE code-signing key. Otherwise, a fresh RSA key pair is generated automatically.

`-f|--file <userdatafile>`

These options specify a file for the `userdata` block.

The option to load a file for the `userdata` block is included only for example purposes.

4.3.8.3. Running the example application

1. Change to the host application directory by running the following command:

Linux

```
cd ~/buildhost/csee/tickets/hostside/
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\hostside\csee\tickets\hostside
```

2. Run the hosttickets example application, specifying the simple key created earlier:

Linux

```
./hosttickets -k simple,ticketkey
```

Windows

```
hosttickets.exe -k simple,ticketkey
```

3. Complete the on-screen instructions:

```
Enter string to be encrypted (256 characters maximum): lskjfdljsdlfjsdlk
HostSide> Loading security world key (simple,ticketkey)

Please present the cardset protecting the key:
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

HostSide> Creating World: init status was 0 (OK)
HostSide> Sending ticket for private RSA key to module
HostSide> Generating AES session key and creating blob under public RSA key
HostSide> Sending key blob to module
HostSide> Sending cipher-text to module
HostSide> decrypted cipher text received from SEE machine:
"lskjfdljsdlfjsdlk"
HostSide> Thank you for watching. The end.
```

4.3.8.4. What the code actually does

The host-side code performs the following tasks in order:

1. It prompts the user for a string.
2. It acquires an RSA key pair, either freshly created or loaded from the Security World (only HSM protected key pairs are supported).
3. It sends a ticket for the private half of the RSA key to the HSM-side code.
4. It generates a session key (DES3).
5. It encrypts the session key as a blob with the public half of the RSA key.
6. It sends the resulting blob to the HSM-side code.
7. It encrypts the string with the session key.
8. It sends the encrypted string to the HSM-side code.
9. It receives the decrypted string back from the HSM.

The HSM-side code awaits jobs from the host and performs the following tasks in order:

1. It receives and redeems the ticket for the private RSA key.
2. It receives the session key blob and decrypts it with the private RSA key.
3. It receives the encrypted string and decrypts it with the session key.
4. It sends the decrypted string back to the HSM.

4.3.9. Example: Benchmark

This example source code implements a very simple utility that uses an SEE machine written in C to time stamp requests to benchmark the speed of response to requests. You can use it for benchmarking during the development of other SEE machines or adapt it as required.

Your SEE-Ready HSM must have an onboard real-time clock for this example to run correctly, and you must have set the clock using the `rtc` command-line utility

This utility does not accept encrypted user data.

4.3.9.1. bm-test

The `bm-test` example application takes the following arguments:

```
bm-test [-l|--log <LOGFILE>][-a|--userdata <userdatafile>] <APPNAME>,<IDENT>
```

`-l|--log <LOGFILE>`

These options specify a file name to which to write time-stamps. If no log file is specified, no logging occurs.

`-a|--userdata <userdatafile>`

These options specify a file for an (optional) `userdata` block.

`<APPNAME>,<IDENT>`

These parameters specify the *APPNAME* and *IDENT* of a key that is to be into the SEE machine (and that SEE machine thereafter uses for signing purposes).

This utility does not have the `--slot` or `--debug` standard options.

4.3.9.2. bm-verify

The `bm-verify` example application takes the following arguments:

```
bm-verify <LOGFILE>
```

The *LOGFILE* parameter specifies the name of the log file created by the `bm-test` example application (specified by that application's `-l|--log` option).

4.3.9.3. Packing and loading the SEE machine

To pack and load the SEE machine:

1. Change to the module directory by running the command:

   For **nShield Solo**

   Linux

   ```
   cd ~/buildSoloMod/csee/benchmark/module
   ```

   Windows

   ```
   cd C:\Users\<USER-NAME>\Documents\buildSoloMod\csee\benchmark\module
   ```

   For nShield XC

   Linux

   ```
   cd ~/buildXCMod/csee/benchmark/module
   ```

Windows

```
cd C:\Users\<USER-NAME>\Documents\buildXCMod\csee\benchmark\module
```

2. Use the `tct2` command line utility to convert the file into a SAR file.

   For **nShield Solo**

   Convert the `bm-machine.sxf` file into a SAR file.

   ```
   tct2 -m 1 --pack --infile bm-machine.sxf --outfile bm-machine.sar
   ```

   For nShield XC

   Convert the `bm-machine.elf` file into a SAR file.

   ```
   tct2 -m 1 --pack --infile bm-machine.elf --outfile bm-machine.sar
   ```

3. Load the SEE machine into the HSM by running the command:

   ```
   loadmache bm-machine.sar
   ```

   This example describes how to load the SEE machine by running the `loadmache` command-line utility. In a production environment, you can choose to configure the `load_seemachine` section of the host or client configuration file so that an SEE machine is loaded automatically. For information about configuration files, see HSM and client configuration files (network-attached HSMs) or Hard-server configuration files (PCIe and USB HSMs).

4. Generate a key for the benchmark application to use by running the command and completing the prompts in the output as follows:

   Linux

   ```
   generatekey simple
   ```

   Output:

   ```
   protect: Protected by? (token, module) [token] >
   recovery: Key recovery? (yes/no) [yes] >
   type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,
   HMACRIPEMD160, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512,
   HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
   size: Key size? (bits, minimum 1024) [2048] >
   OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
   ```

```
>
ident: Key identifier? [] > benchmark-test
plainname: Key name? [] > benchmark
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, nvrammachine, a3a8userdata, hellomachine, rtccode) []
>
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
operation     Operation to perform              generate
application   Application                       simple
protect       Protected by                      token
slot          Slot to read cards from           0
recovery      Key recovery                      yes
verify        Verify security of key            yes
type          Key type                          RSA
size          Key size                          2048
pubexp        Public exponent for RSA key (hex)
ident         Key identifier                    benchmark-test
plainname     Key name                          benchmark
seeintegname  SEE integrity key
nvram         Blob in NVRAM (needs ACS)         no

Loading `ocs-dev':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: `ocs-dev' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: /opt/nfast/kmdata/local/key_simple_benchmark-test
```

Windows

```
generatekey -m 1 simple
```

Output:

```
protect: Protected by? (token, module) [token] >
recovery: Key recovery? (yes/no) [yes] >
type: Key type? (AES, DES2, DES3, DH, DHEx, DSA, EC, ECDH, ECDSA, Ed25519,
                 HMACRIPEMD160, HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512,
                 HMACTiger, KCDSA, Rijndael, RSA, X25519) [RSA] >
size: Key size? (bits, minimum 1024) [2048] >
OPTIONAL: pubexp: Public exponent for RSA key (hex)? []
>
ident: Key identifier? [] > benchmark-test
plainname: Key name? [] > benchmark-test
OPTIONAL: seeintegname: SEE integrity key?
(a3a8machine, a3a8userdata, hellomachine, rtccode) [] >
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation     Operation to perform              generate
 application   Application                       simple
 protect       Protected by                      token
 slot          Slot to read cards from           0
 recovery      Key recovery                      yes
 verify        Verify security of key            yes
 type          Key type                          RSA
 size          Key size                          2048
 pubexp        Public exponent for RSA key (hex)
 ident         Key identifier                    benchmark-test
 plainname     Key name                          benchmark-test
 seeintegname  SEE integrity key
```

```
 nvram         Blob in NVRAM (needs ACS)          no

Loading 'dev-ocs':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: 'dev-ocs' #1
 Module 1 slot 0:- passphrase supplied - reading card
Card reading complete.

Key successfully generated.
Path to key: C:\ProgramData\nCipher\Key Management Data\local\key_simple_benchmark-test
```

4.3.9.4. Running the example application

To use the example change to the host application directory by running the command:

Linux

```
cd ~/buildhost/csee/benchmark/hostside/
```

Windows

```
cd C:\Users\<USER-NAME>\Documents\host\csee\benchmark\hostside
```

Run the bm-test example application as follows, completing any on-screen instructions in the output:

Linux

```
./bm-test -l bmtest.log simple benchmark-test
```

Windows

```
bm-test.exe -l bmtest.log simple benchmark-test
```

Output:

```
Please insert the next operator card and press enter.
Please enter card passphrase:
1 878 878.00
2 1758 879.00
3 2639 879.67
4 3522 880.50
5 4406 881.20
6 5284 880.67
.
.
.
```

The application will run indefinitely, the user must terminate the applica tion manually by using **Ctrl-C**.

Run the bm-verify example application, specifying the log file, `bm-test.log`, created in the previous step by the `bm-test` application.

Linux

```
./bm-verify bmtest.log
```

Windows

```
bm-verify.exe bmtest.log
```

Output:

```
Verified timestamp #1.
Verified timestamp #2.
Verified timestamp #3.
Verified timestamp #4.
Verified timestamp #5.
Verified timestamp #6.

.
.
.
```

4.3.9.5. What the code actually does

The host program performs the following tasks in order:

1. It tickets a generated key into the SEE machine.
2. The SEE machine uses that key for signing purposes.
3. Each request is concatenated with the current time and then signed.
4. The signature is concatenated with the time and then returned to the host side.

On the host side, two programs are generated:

- `bm-test`
- `bm-verify`.

The `bm-test` command is used to generate pseudo-random values that are sent to the HSM-side code to be signed. Every second, the total number of completed time-stamp requests is printed, along with the average number completed each second.

The `bm-verify` command looks for the file specified as *LOGFILE* on the host. From this file, `bm-verify` extracts the public key and verifies the time-stamp requests until it finds an invalid request or reaches the end of the file.

4.3.9.6. About the Java example

We supply a Java version of the `HelloWorld` example. This consists of the source files for host-side applications that you can run with the example SEE machines written in C (or any other SEE machines written in any language) in order to understand how simple SEE machines work.

For information about the C examples for SEElib, see Examples for SEElib

The Java SEE example files can be found within the `nCipherKM-SEE-Examples jar` located in `/opt/nfast/java/examples` (**Linux**) or `%NFAST_HOME%\java\examples` (**Windows**). A common directory is also supplied which contains files that are used by more than one of the examples.

The Java examples have the same options as their equivalent, similarly named C examples.

4.3.9.6.1. Supported versions of Java

The following versions of Java have been tested to work with, and are supported by, your nShield Security World Software:

- Java8 (or Java 1.8x)
- Java11
- Java17
- Java21

We recommend that you ensure Java is installed before you install the Security World Software. The Java executable must be on your system path.

If you can do so, please use the latest Java version currently supported by Entrust that is compatible with your requirements. Java versions before those shown are no longer supported. If you are maintaining older Java versions for legacy reasons, and need compatibility with current software, please contact https://www.oracle.com/java/technologies/javase-jdk11-downloads.html for Java downloads.

4.3.9.6.2. HelloWorld.java

The `HelloWorld.java` example is *not* intended to be the basis for any real world applications. It is intended only to demonstrate host-side use of an SEE machine by code written in Java.

First, ensure you have already built the file `hello.sxf` as described in Examples for SEElib

converted this into the file `hello.sar` and loaded it into the HSM as described in Signing, packing, and loading the SEE machine.

To build the example:

1. Change to the example directory by running the command:

   Linux

   ```
   cd /opt/nfast/java/examples
   ```

   Windows

   ```
   cd %NFAST_HOME%\java\examples
   ```

2. Extract the example files by running the command:

   Linux

   ```
   jar xf nCipherKM-SEE-Examples.jar
   jar xf ../classes/nCipherKM-jhsee.jar
   ```

   Windows

   ```
   jar xf nCipherKM-SEE-Examples.jar
   jar xf ..\classes\nCipherKM-jhsee.jar
   ```

3. Compile the example using this command:

   Linux

   ```
   javac -cp /opt/nfast/java/classes/nCipherKM.jar com/ncipher/see/hostside/*.java
   javac -cp
   .:/opt/nfast/java/classes/nCipherKM.jar com/ncipher/see/hostside/examples/helloworld/HelloWorld.java
   ```

   Windows

   ```
   javac -cp "%NFAST_HOME%\java\classes\nCipherKM.jar com\ncipher\see\hostside\*.java"
   javac -cp "%NFAST_HOME%\java\classes\nCipherKM.jar ^
   com\ncipher\see\hostside\examples\helloworld\HelloWorld.java"
   ```

To run the `helloworld` example:

1. Ensure you are in the example's directory by running the command:

   Linux

   ```
   cd /opt/nfast/java/examples
   ```

Windows

```
cd %NFAST_HOME%\java\examples
```

2. Run the example:

Linux

```
java -cp .:/opt/nfast/java/classes/nCipherKM.jar
com/ncipher/see/hostside/examples/helloworld/HelloWorld <FILENAME> [<USERDATA>]
```

Windows

```
java -cp "%NFAST_HOME%\java\classes\nCipherKM.jar ^
com\ncipher\see\hostside\examples\helloworld\HelloWorld <FILENAME> [<USERDATA>]"
```

In this example, <FILENAME> is the name of an input file to pass to the SEE machine as an SEE job, and <USERDATA> the name of an (optional) userdata file. The SEE machine transforms the input by replacing all lowercase alphabetic characters in <FILENAME> with their uppercase equivalents and replacing any characters in <FILENAME> that are also found in <USERDATA> (if supplied) with the character X.

5. Debugging SEE machines

This chapter provides some guidance on debugging an SEE machine.

5.1. Debugging settings and output

To debug an SEE application effectively, you must have:

- Enabled SEE debugging when creating the Security World in which the application is to run, see new-world (`dsee` and `dseeall` options).
- Set `Cmd_CreateSEEWorld_Args_flags_EnableDebug` when creating the SEE World.

If you try to set the `Cmd_CreateSEEWorld_Args_flags_EnableDebug` flag in a Security World that does not allow SEE debugging, the `CreateSEEWorld` command returns `AccessDenied`. This also occurs if you call `CreateSEEWorld` in a Security World where SEE debugging is restricted and an appropriate certifier is not present.

5.1.1. Debugging authorization

Access to the SEE trace buffer is controlled by the Security World in which the SEE machine runs. Every Security World has exactly one of the following properties:

- Restricted SEE debugging

  This is the default setting. When SEE debugging is restricted, there is no delegation key from K_{NSO} for accessing the SEE trace buffer. All Security Worlds created by software released before the introduction of SEE have restricted SEE debugging. A full quorum of Administrator Cards is required to access the SEE trace buffer in such Security Worlds.

- Authorized SEE debugging

  In this case, a delegation key from K_{NSO} exists to allow access to the SEE trace buffer. A subset of a full quorum of the Administrator Cards is required to access the SEE trace buffer in such Security Worlds. This delegation key must have been created and the number of cards required to authorize access to the SEE trace buffer must have been specified when the Security World was created.

- No access-control SEE debugging

  In this case, no authorization of any kind is required for accessing the SEE trace buffer.

No cards are required to access the SEE trace buffer in such Security Worlds. This property must have been specified when the Security World was created.

5.1.2. Obtaining debugging output

For SEE machines that require support from a host-side `see-*-serv` utility, you can run the `see-*-serv` utilities with the `--trace` or `--plain-trace` option to perform tracing automatically.

For SEE machines using the `SEElib` architecture, the `TraceSEEWorld()` command can be used to return debugging information. An example of this is provided in the `a3a8` host-side example code. See A3A8 example.

Data written to standard output and standard error on the HSM is written to the SEE World's Trace Buffer. The Trace Buffer is a 3000 character circular buffer: if more than 3000 characters are written to it without being retrieved, information is lost on a first-in/first-out basis. The `TraceSEEWorld` command retrieves the contents of the buffer so that the host can analyze or display them.

If the SEE machine crashes, a SEE register dump is printed to the SEE Trace Buffer for the nShield Solo, but not for the nShield Solo XC.

For example, assume that the HSM code calls the following command:

```
printf("Hello World!\n");
```

The string `Hello World!\n` is pushed into the Trace Buffer. A host-side call to `TraceSEEWorld` would then return this string and empty the buffer.

If a SEE World is terminated by the HSM (for instance, if its last remaining thread exits or it causes a fatal signal to be raised), a diagnostic message is usually sent to the Trace Buffer to help debug the problem.

5.1.2.1. Example Debug

If an illegal access violation (segmentation fault) occurs, the tail of the Trace Buffer looks similar to this:

```
   *** World exits: thread 28 caused CPU exception
DSI exception:
Exception vector 00300h
   r0 =001D9E40h r1 =001D9F38h r2 =00C4E090h r3 =00000008h
   r4 =00000000h r5 =00C00444h r6 =00000000h r7 =001C21B1h
   r8 =00C39CB8h r9 =00000019h r10=40000000h r11=00002000h
   r12=00000000h r13=00D08048h r14=00000000h r15=00000000h
```

```
   r16=00000000h r17=00000000h r18=00000000h r19=00000000h
   r20=00000000h r21=00000000h r22=00000000h r23=00C40000h
   r24=FFFC5CD0h r25=00C3A750h r26=00C40000h r27=00C40000h
   r28=00000000h r29=00000000h r30=00000000h r31=00D00000h
   XER=20000000h CR =20000000h LR =00C00444h CTR=00C39B9Ch
   PC =00C00448h MSR=0000F030h
   f0 =0000000000000000h f1 =0000000000000000h
   f2 =0000000000000000h f3 =0000000000000000h
   f4 =0000000000000000h f5 =0000000000000000h
   f6 =0000000000000000h f7 =0000000000000000h
   f8 =0000000000000000h f9 =0000000000000000h
   f10 =0000000000000000h f11 =0000000000000000h
   f12 =0000000000000000h f13 =0000000000000000h
   f14 =0000000000000000h f15 =0000000000000000h
   f16 =0000000000000000h f17 =0000000000000000h
   f18 =0000000000000000h f19 =0000000000000000h
   f20 =0000000000000000h f21 =0000000000000000h
   f22 =0000000000000000h f23 =0000000000000000h
   f24 =0000000000000000h f25 =0000000000000000h
   f26 =0000000000000000h f27 =0000000000000000h
   f28 =0000000000000000h f29 =0000000000000000h
   f30 =0000000000000000h f31 =0000000000000000h
   FPSCR=00000000h
```

The program counter, which is currently at position `00C00448h` in the PowerPC-based compilation shows where this access occurs.

The following excerpt from the PowerPC based map file created at application link time (by specifying the `-map` option to the linker) indicates that the problem address is in `main.o`:

```
.text 0x00c00000                0x3a0ac
  *(.text.stub.text.*.gnu.linkonce.t.*)
    .text 0x00c00000            0xa5c usermain.o
          0x00c00160            main
    .text 0x00c00a5c            0x544 .\lib-ppc-gcc\seelib.a(nfstrerr.o)
          0x00c00a5c            NFast_StrError
```

To find out which instruction is causing the segmentation fault, calculate the offset into `main.o`. The formula is:

```
program_counter - object_base_address
```

The calculation is as follows:

```
00C00448h -
00C00000
--------
0x00448h
```

Once the location of the problem is located in this way, investigate it as follows:

1. Recompile the source with the `-g` option and no optimization (if you did not originally compile it with these options).

2. Run an object dump utility on the object files `powerpc-codesafe-linux-gnu-objcopy`.

The head of the generated object is now similar to the following for PowerPC based objects:

```
434:    38 7a 03 34   addi    r3,r26,820
438:    38 80 00 08   li      r4,8
43c:    4c c6 31 82   crclr   4*cr1+eq
440:    48 00 00 01   bl      440 <main+0x2e0>
444:    38 60 00 08   li      r3,8
448:    80 03 00 00   lwz     r0,0(r3)
44c:    4b ff fe 74   b       2c0 <main+0x160>
450:    3c 80 00 00   lis     r4,0
```

From this output is it possible to see that the segmentation fault is caused by an illegal access to the pointer held in `R4` (which the register dump showed to be `80000004h`, an obviously invalid user mode memory address). The source shows plainly that the instruction at offset `0458h` in `usermain.o` is trying to assign to `*i`, but `i` has not been allocated. The bug can now be fixed and the program rebuilt.

5.2. Finding memory leaks with stattree

You can use the `stattree` command-line utility to find memory leaks. Run the command:

Linux

```
stattree | grep Mem
```

Windows

```
stattree | find "Mem"
```

For each HSM in the Security World, this command produces output that reports values for the total memory (`MemTotal`), the memory currently allocated to the kernel (`MemAllocKernel`), and the memory currently allocated to the loaded SEE machine (`MemAllocUser`).

If no SEE machine is loaded, the output from this `stattree` command (if there is only one HSM) looks similar to the following:

```
-MemTotal            128921600
-MemAllocKernel      1355776
-MemAllocUser        0
```

If an SEE machine is loaded, the output from this `stattree` command (if there is only one HSM) looks similar to the following:

```
-MemTotal            128921600
-MemAllocKernel      1355776
-MemAllocUser        1032192
```

You can monitor a loaded SEE machine's memory usage by either repeatedly running and checking output from `stattree` or by writing code to call the nCore statistics APIs directly. In any case, if any reported memory value appears to being growing continuously over time, this probably indicates some kind of memory leak.

5.3. Segment addresses for Solo

SEE executables are non-relocatable; that is, they are loaded in memory at the addresses specified in the image. Ensure that you choose these addresses carefully so that they map onto usable RAM and do not overlap with memory being used by the kernel. Typically, this means you must choose an address at the high end of RAM.

Different HSM types have different mappable memory ranges.

- The CodeSafe compiler sets all values for Solo XC and later HSM models.
- You have to set the ranges in the CodeSafe application code if you are developing for Solo +.

  The rest of this section describes guidelines for Solo +.

To determine your HSM type, run the `enquiry` command-line utility and check the `SEE Machine Type` output. You can then determine where the mappable memory range starts from this table:

`SEE Machine Type`	**Start of mappable range**
`PowerPCSXF`	0x00400000

These ranges follow the approximately 4MB of RAM reserved for use by the kernel.

You can use the `stattree` command-line utility to find the total length of the mappable range. Run the command:

Linux

```
stattree | grep MemTotal
```

Windows

```
stattree | find "MemTotal"
```

This command produces output that reports values for the total memory (`MemTotal`) for each HSM in the Security World.

For Solo +, we recommend the following segment addresses as starting points:

SEE Machine Type	PowerPCSXF
`text` segment start	`0xa00000`
`data` segment start	`0x00d00000`
Arguments to the linker	`-Ttext 0xa00000 -Tdata 0xd00000`

For large SEE machines more space may be needed in the text segment, causing a linker error of the following form:

```
powerpc-codesafe-linux-gnu-ld: section .data [00d00000 -> 00d0327f] overlaps section .text [00c00000 -> 00d7bd8b]
powerpc-codesafe-linux-gnu-ld: section .sdata [00d03280 -> 00d035ef] overlaps section .text [00c00000 ->
00d7bd8b]
powerpc-codesafe-linux-gnu-ld: section .sbss [00d035f0 -> 00d036ab] overlaps section .text [00c00000 -> 00d7bd8b]
powerpc-codesafe-linux-gnu-ld: section .bss [00d036b0 -> 00d0854f] overlaps section .text [00c00000 -> 00d7bd8b]
```

To resolve this example error, you could move the data segment start point upward (for example, to `0x00e00000`) as necessary to prevent the overlap. Alternatively (or additionally), you could move the text segment start point downward.

5.4. Vulnerability test harness

We supply a test harness called `vulnerability.o` that can be used for debugging SEE machines. It supplies a standard set of command-line arguments and environment variables to the SEE environment, as well as providing the standard `stdioe` and socket support.

Because the `vulnerability.o` test harness is insecure, we recommend that you *not* link `vulnerability.o` into a production SEE machine.

5.5. Troubleshooting guide

Symptom	Possible problems	Solution
`SEEJob` takes a long time then fails with `HardwareFailed`	The SEE machine has deadlocked or entered an infinite loop which prevents the job from returning and causes the `SEEJob` to trigger the command time-out.	Check the code for possible deadlocks or infinite loops. Non-obvious problems can be debugged by writing progress reports to the Trace Buffer and calling `TraceSEEWorld` after the job returns `HardwareFailed`.
`CreateSEEWorld` fails with `BadMachineImage`	No SEE machine is loaded.	Load an SEE machine
SEE machine loading fails with `BadMachineImage`	The file being loaded is not a correctly formatted SAR file.	Ensure that the correct SEE machine file is being loaded. Ensure that the SEE machine has been properly processed by the Trusted Code Tool into a SAR file.
	The SEE machine file is corrupted.	Rebuild the SEE machine, or revert to a known good back-up.
	The SEE machine has been compiled or linked with the wrong options.	SEE machines must be nonexecutable, uncompressed, non-relocatable AIFs or SXFs, packaged as SAR files.
`CreateSEEWorld` fails with `InvalidCertificate`	The machine signing hash on `userdata` signatures does not match any signature hash on the currently loaded machine.	Ensure the correct SEE machine with the correct signatures is loaded. Ensure the correct user data is being passed to `CreateSEEWorld`. Ensure the user data signatures are correct.
SEE machine loading fails with `InvalidCertificate`.	The SEE machine signatures were created incorrectly.	SEE machine signatures must be created with the machine key specification `--is-machine`. Recreate the SEE machine SAR with correct signatures.
The SEE machine crashes, and Trace Buffer output shows raised signal.	Dependent on signal number.	Check `stdh.h` and `signal.h` for signal descriptions then check the code to see how that signal could be raised.

Symptom	Possible problems	Solution
`AccessDenied` from `CreateSEEWorld`.	SEE World debugging is not available in Security World.	Check the Security World's SEE debugging policy.
	SEE machine is returning `AccessDenied` in `SEElib_initComplete`.	Check the SEE machine set-up code to see where it might be passing `AccessDenied` to `SEElib_initComplete`, and fix the cause of that, if necessary.
All `SEEJob`s return with `Status_Cancelled`.	`SEElib` transaction listener is not running.	If you are using `SEElib_Transact` you must call `SEElib_StartTransactListener` before making use of `SEElib_Transact`.
`NoModuleMemory` is returned from the `CreateSEEWorld` command.	Segment addresses clash with kernel pages.	Adjust segment positions away from kernel RAM; see Segment addresses for Solo .
	Segment addresses overlap.	Adjust segment away from each other; see Segment addresses for Solo .
	Segment addresses are not usable RAM.	Adjust segment positions to usable RAM; see Segment addresses for Solo .
`NoModuleMemory` is returned when loading a SEE machine.	Userdata has been specified but is not expected.	Exclude the userdata.
	The previous SEE machine has not been cleared	Clear the previous SEE machine; see clearing a SEE machine from the front panel or clearing a SEE machine remotely
Error from link: `section .data [hhhhhhhh → hhhhhhhh] overlaps section .text [hhhhhhhh → hhhhhhhh]`	Segment addresses overlap.	Adjust segment away from each other; see Segment addresses for Solo .

6. Deploying SEE Machines

This chapter discusses the deployment of SEE machines after their development is complete. It includes information about Feature Enabling as this applies to SEE.

Deploying a SEE machine involves the following steps:

1. Sign and encrypt the SEE machine. See Signing methods and Encryption.
2. Obtain an export certificate for the SEE machine from Entrust and incorporate the certificate in the distribution. See Obtaining and using export certificates.
3. Distribute the SEE machine to customers.

6.1. About the Feature Enabling Mechanism (FEM)

Entrust provides a Feature Enabling Mechanism (FEM) that controls the software that any given HSM can use. This is used to control access to the `SetSEEMachine` command that loads the SEE machines.

The `SetSEEMachine` command can be authorized in either of the following ways:

- The `GeneralSEE` static feature is set with a bit in the EEPROM. If this bit is set, the command can load any SEE machine without further certificates or authorization.
- If the `GeneralSEE` static feature is not applied, the command requires a dynamic Feature Enabling certificate chain to load a SEE machine.

All CodeSafe *development* environments have the `GeneralSEE` static feature. However, to deploy an already-developed SEE machine, you require the dynamic Feature Enabling certifi cate chain.

Customers who require the dynamic certificate chain can load a SEE machine only when the key used to sign the SEE machine is export approved by Entrust through the provision of a signing certificate (an ADDER certificate). See Obtaining and using export certificates.

The SEE machine signing (ADDER) certificate authorizes `SetSEEMachine` on any HSM, but the dynamic Feature Enabling certificate chain is valid only on the specified HSM.

6.2. Obtaining and using export certificates

You must understand and agree to the conditions for exporting SEE machines. Contact Entrust for full details of these conditions.

To obtain an export certificate for a SEE machine:

Users with a Restricted SEE, [SEE(R)], enabled Connect will need to run `update world files` to pull the ADDER cert onto the Connect file system to load a SEE machine.

1. Generate a signing key and send the hash to Entrust together with a description of the SEE machine.

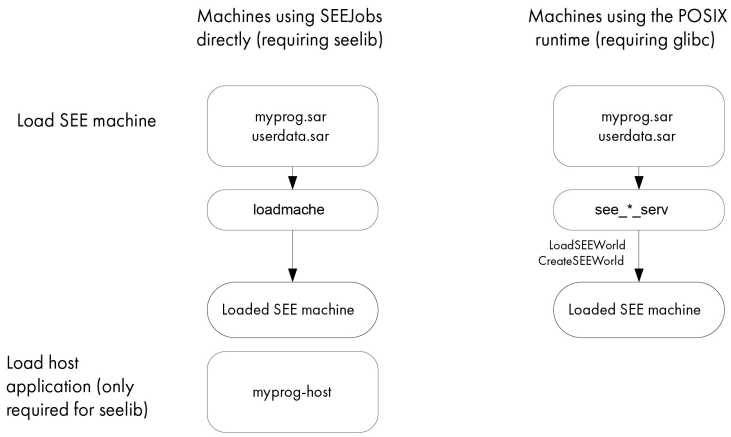
   Entrust approves the SEE machine for export and sends you an ADDER certificate to allow the SEE machine signed by the specified key to run.

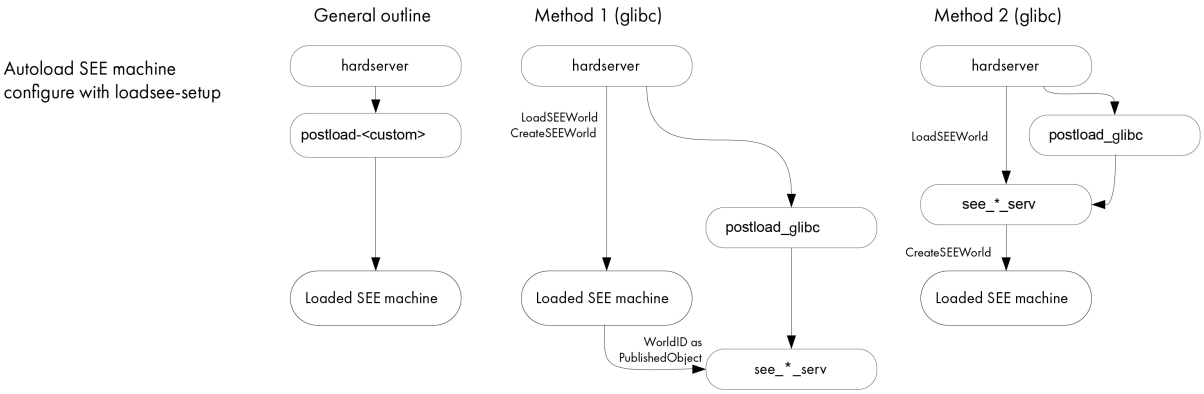
2. Sign the SEE machine with the signing key supplied to Entrust and, optionally, encrypt it.
3. Develop an installation process that places the certificate in the `/opt/nfast/femcerts` (Linux systems) or `%NFAST_CERTDIR%` (Windows) directory.
4. Distribute the signed SEE machine and the certificate to end-users with the appropriate installation instructions.

6.3. Automatically loading a SEE machine

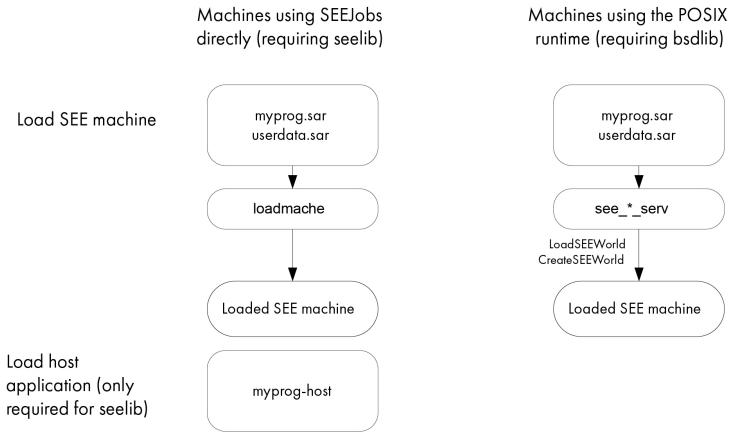
The figures below outline different methods for loading a SEE machine.

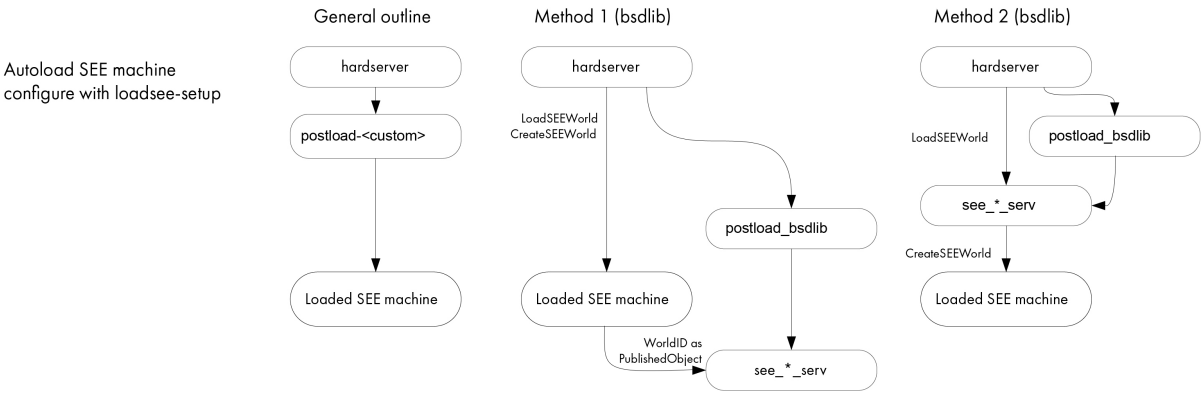
Loading SEE machines for Solo XC:





Loading SEE machines for Solo PCIe:





You can load SEE machines manually by running the `loadmache` command-line utility or, optionally, you can load SEE machines that require support from a host-side `see-*-serv` utility by specifying the `-M` option when you run the utility.

However, you can also configure the hardserver to load SEE machines automatically whenever the HSM is initialized (that is, when the hardserver starts, or restarts, or after the HSM receives a `ClearUnit` command).

To configure the hardserver to load a SEE machine automatically, you must edit the settings in the host systems hardserver configuration file. Entrust provides the `loadsee-setup` command-line utility to help you set up, display, and remove settings in the hardserver configuration file that control the automatic loading of SEE machines.

For a usage description of the `loadsee-setup` command-line utility, see Loadsee setup. For more information about the configuration files, see HSM and client configuration files (network-attached HSMs) or Hardserver configuration files (PCIe and USB HSMs)

The `loadsee-setup` utility configures the hardserver settings that specify:

- The name of the SEE machine file to be automatically loaded
- If appropriate, the name of an accompanying `userdata` file.
- If appropriate (if `userdata` is specified), the published-object name for the SEE machine
- The name of an appropriate post-load program (to perform setup and initialization tasks for the SEE machine) and any necessary arguments for it (a `-m` option to specify an HSM is automatically added)

For SEE machines that require support from a host-side `see-*-serv` utility, Entrust provides the `postload-bsdlib` post-load program, which runs the appropriate host utility, in restricted mode, while returning control back to the hardserver. The `postload-bsdlib` program takes the same arguments as the `see-*-serv` host utilities (see see-*-serv utilities), together with a `--provision` argument that takes one of the following parameters to specify which utility to run:

- `stdoe`
- `stdioe`
- `sock`
- `stdioesock`

For SEE machines using the `SEElib` architecture, it is usually necessary to write a custom post-load program.

6.3.1. Automatically loading a glibc SEE machine with userdata

To configure the hardserver configuration file to automatically load a `glibc` SEE machine and its accompanying `userdata` file, run a command similar to the following example:

Linux

```
loadsee-setup -m1 -M /tmp/MySEEMachine.sar -U /tmp/MyUserdata.sar -p MyPublishedObjectName -P glibc -A "--
provision sock -p MyPublishedObjectName"
```

Windows

```
loadsee-setup -m1 -M C:\MySEEMachine.sar -U C:\MyUserdata.sar -p MyPublishedObjectName -P glibc -A "--provision
sock -p MyPublishedObjectName"
```

In this example, `MySEEMachine.sar` is the SEE Machine (packed as a SAR file), `MyUserdata.sar` is the `userdata` (packed as a SAR file), `MyPublishedObjectName` is the name to use for publishing the `KeyID` of the started SEE machine, and the `glibc` parameter specifies use of the `postload-bsdlib` post-load program.

The `sock` parameter in this example tells `postload-bsdlib` to run the `see-sock-serv` host utility. If a different host utility were necessary, you would specify the appropriate parameter for the appropriate utility (`stdoe`, `stdioe`, or `stdioesock`).

When running a command of this form, ensure that the parameters specifying name of the published object (in this example, `MyPublishedObjectName`) are the same for both the `loadsee-setup` utility and the `postload-bsdlib` program.

For more information about the `loadsee-setup` command-line utility, see Loadsee setup.

6.3.2. Automatically loading a glibc SEE machine without userdata

To configure the hardserver configuration file to automatically load a `glibc` SEE machine without its accompanying `userdata` file (which instead is to be loaded by the host utility), run a command similar to the following example:

Linux

```
loadsee-setup -m1 -M /tmp/MySEEMachine.sar -P glibc -A "--provision sock --userdata-sar /opt/nfast/nc-
seemachines/MyUserdata.sar"
```

Windows

```
loadsee-setup -m1 -M C:\MySEEMachine.sar -P glibc -A "--provision sock --userdata-sar C:\MyUserdata.sar"
```

In this example, `MySEEMachine.sar` is the SEE Machine (packed as a SAR file) and the `glibc` parameter specifies use of the `postload-bsdlib` post-load program.

The `sock` parameter in this example tells `postload-bsdlib` to run the `see-sock-serv` host utility. If a different host utility were necessary, you would specify the appropriate parameter for the appropriate utility (`stdoe`, `stdioe`, or `stdioesock`).

The `MyUserdata.sar` parameter in this example, passed to the `postload-bsdlib` program, specifies a `userdata` file (packed as a SAR) that is to be loaded by the host utility.

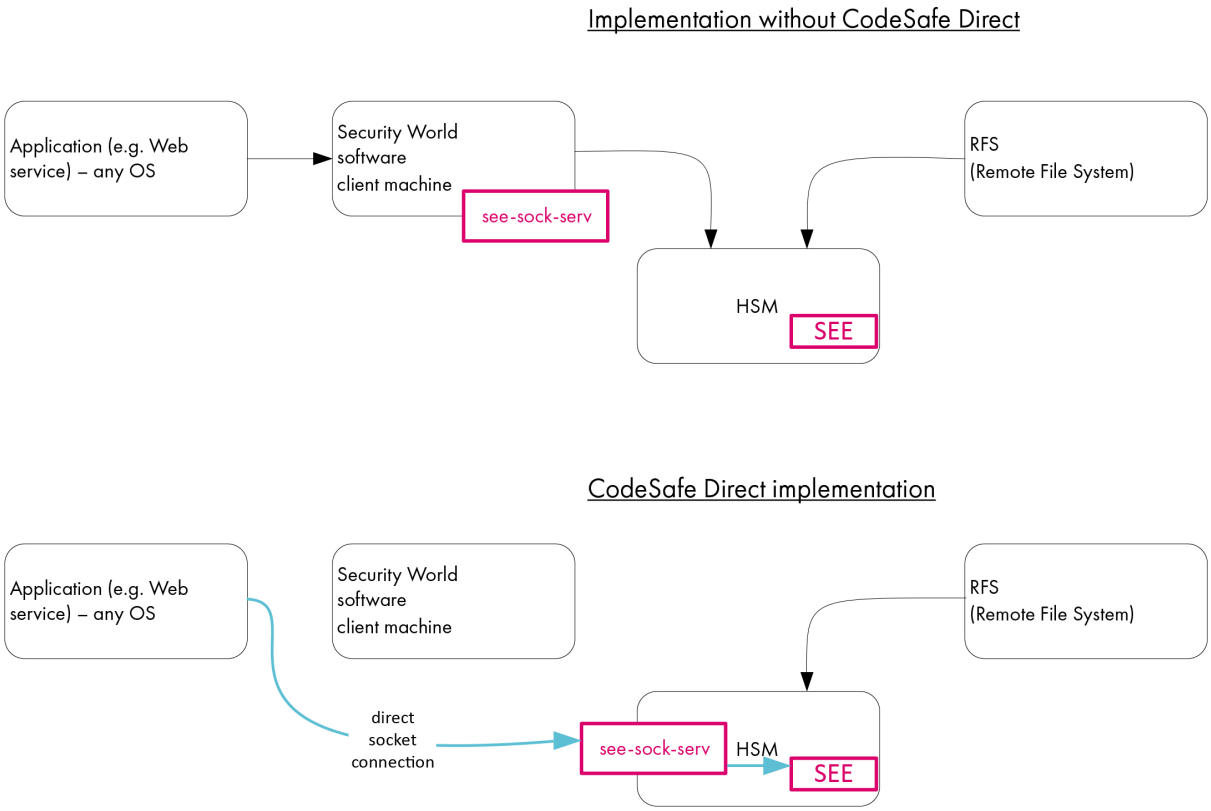
To specify `userdata` that has not been packed as a SAR file, use the `--userdata-raw` option instead of `--userdata-sar`.

To turn on SEE debugging, pass one of the options `--trace` or `--plain-trace` as an argument for the post-load program. See also Debugging SEE machines.

The host utility will be run in restricted mode, using the `-r` option.

6.4. Configuring the nShield Connect to use CodeSafe Direct

The CodeSafe client can be any OS platform (including mainframe, Non-Stop or embedded device). The use of CodeSafe Direct eliminates proxy devices, complexity and points of failure.



The nShield Connect can be configured to receive direct socket connections from the SEE machine via `see-sock-serv`, removing the need for a client machine. You do this by specifying `postload_prog` and `postload_args` in the `load_seemachine` section of the nShield Connect hardserver configuration file, located in `NFAST_KMDATA/hsm-<ESN>`, where *<ESN>* is the Electronic Serial Number of the HSM. (For more information about this section of the configuration file, see load_seemachine.

CodeSafe Direct is supported on glibc-based SEE machines only: the functionality is not available on SEElib-based machines.

The configuration file can be managed in two ways: via the front panel of the nShield Connect (see Configuring a SEE machine using the front panel), and by using the remote configuration functions to push a `config.new` file, containing the `postload_prog` and `postload_args` settings, to the HSM.

For more information, see HSM and client configuration files.

6.5. Configuring a SEE machine using the front panel

To use `see-sock-serv` directly, you must create a glibc SEE machine.

Ensure that the SEE machine for the application is in the `/opt/nfast/custom-seemachines` (**Linux**) or `%NFAST_HOME%\custom-seemachines` (**Windows**) directory on the remote file sys-

tem.

If a SEE machine has previously been loaded on a network-attached HSM with a front panel, such as an nShield Connect, clear the current SEE machine from memory in one of the following ways:

- Press the **Clear** button on the front of the HSM.
- Log in to a host machine as a user in the `nfast` group and run the following command (*m1* is the Security World's module number for the HSM whose front panel you used in the previous steps):

```
sudo /opt/nfast/bin/nopclearfail -c -w -m1
```

6.5.1. Configuring a glibc SEE machine

Select the CodeSafe menu option, and enter the following information when prompted:

1. The name of the SEE machine file.
2. The name of the userdata file.

For CodeSafe Direct, the `userdata` file must be packed as a SAR file.

3. The type of custom SEE machine you are using (`BSDlib sockserv`). `worldid_pubname`, `postload_prog`, and `postload_args` will be passed to `load_seemachine`. For detailed descriptions of the options in this section, see load_seemachine.

6.5.2. Configuring a SEElib SEE machine

Select the CodeSafe menu option, and enter the following information when prompted:

1. The name of the SEE machine file.
2. The name of the userdata file, if required.

   The `userdata` file must be packed as a SAR file.

3. The type of custom SEE machine you are using (`SEElib`).
4. The ID of the SEE World to create.

6.6. Remotely loading and updating SEE machines

The SEE remote push facility allows the remote deployment of CodeSafe SEE machines to an nShield Connect, negating the need to physically visit the HSM to load or update the SEE machine. This is achieved by editing the configuration file on the RFS for a specific nShield Connect to specify the new SEE machine, then setting a configuration flag in the config file to `true`.

Before configuring an HSM to autonomously run a SEE machine and accept updates using the RFS, that HSM must first be set up to accept remotely-pushed configurations. Refer to Remote Administration v13.6.11 User Guide for more information.

For more information about configuring log file storage options, see Configuring log file storage.

Both `SEElib` and `BSDlib sockserv` SEE machines are supported on the nShield Connect.

To configure an nShield Connect to autonomously run a SEE machine and accept updates using the RFS:

1. Place the SEE machine in the following location:
   - **Linux:** `/opt/nfast/custom-seemachines`
   - **Windows:**`C:\Program Files\nCipher\nfast\custom-seemachines`
2. Copy the existing config file to a new file called `config.new`.
3. In the `load_seemachine` section of the `config.new` file for the remote HSM, add or amend the following settings:

   ```
   module=1
   pull_rfs=yes
   machine_file=mymachinename.sar
   userdata=myuserdata.sar
   worldid_pubname=publ_name
   ```

   These settings specify the type, name and user data of the SEE machine you wish to load. For more information about each setting, see load_seemachine.

   For CodeSafe Direct, the `userdata` file must be packed as a SAR file.

   The remote HSM will load the new SEE machine in place of any existing SEE machine. If no `machine_file` value is set, then pushing the config file will remove any existing machines on the HSM.

4. In the `sys_log` section of the `config.new` file for the remote HSM, add or amend the fol lowing settings:

```
[sys_log]
behaviour=push
push_interval=1
```

This allows the HSM to push its `hardserver.log` to the RFS every minute (`push_interval=1`). This change is recommended for troubleshooting and verification purposes. The default is `60` (minutes).

These settings control how and where log messages are written. Using the example above, messages will be written to the `system.log` and `hardserver.log` files of the HSM, which are accessible using the remote file system. You may wish to revise the `push_interval` to a higher value once the nShield Connect has successfully loaded the new SEE machine.

5. Run nopclearfail to clear the module.
6. Run enquiry to check that the module is ready.
7. From the location of the HSM config file, run cfg-pushnethsm to push the new config file to the HSM:

   ```
   cfg-pushnethsm --address=module_IP_address config.new
   ```

   Location:

   - **Linux:** `/opt/nfast/kmdata/hsm-####-####-###/config`
   - **Windows:** `C:\ProgramData\nCipher\Key Management Data\hsm-####-####-###\config`
8. Run `nopclearfail -c -w`.
9. If you are loading a new or different SEE machine, search the HSM's hardserver log for the string `hsc_loadseemachine` to check whether the SEE machine loaded or whether it reported an error.
10. Verify the SEE machine has loaded by running stattree:

    ```
    stattree PerModule 1 ModuleEnvStats
    ```

    A non-zero `MemAllocUser` value indicates that the SEE machine is loaded.

You can do this on any working client, including the RFS if it is also a client, of the nShield Connect. On "XC" HSMs, this requires a firmware version of 12.50.2 or greater.

The HSM pushes the config file back to the RFS with changes:

- The `pull_rfs` flag is set to `no`, because the SEE machine is now loaded.
- The `machine_file` and `userdata` values are now set to the paths to their respective locations in the embedded OS of the HSM.

To load a new SEE machine to multiple nShield Connects, we recommend scheduling down time for each HSM, upgrading them on a per HSM basis. Each nShield Connect configuration file is specific to an individual HSM and each configuration file should be updated separately to load the new SEE machine.

7. Utilities

Entrust supplies the following SEE-specific nShield command-line utilities:

- `elftool`.
- `loadmache`.
- `loadsee-setup`.
- `hsc_loadseemachine`
- The `see-*-serv` host-side utilities:
  - `see-sock-serv`.
  - `see-stdoe-serv`.
  - `see-stdioe-serv`.
  - `see-stdioesock-serv`.
- `seessl-migrate.py`.
- `tct2` (the Trusted Code Tool)

This appendix also describes the following general nShield command-line utilities:

- `nfkmverify`

For a list of all supplied nShield utilities, see nShield v13.6.11 Utilities Reference.

7.1. cpioc

The `cpioc` command-line utility takes a collection of files and packs them up into a `userdata` archive file that the SEE machine can use.

7.1.1. Usage

```
cpioc userdata.cpio <MyFile1> <MyFile2> <MyFile3> <[...]>
```

In this command, `<MyFile1>`, `<MyFile2>`, and `<MyFile3>` represent the files being packed into the `userdata.cpio` file that is generated by the command. You can specify as many files as appropriate.

You can also specify one or more directories; the command automatically packs their contents (including any subdirectories) into the generated `userdata.cpio` file.

7.2. elftool

The `elftool` command-line utility lets you convert ELF format executables into a format suitable for loading as an SEE machine.

7.2.1. Usage

```
elftool [<options>] <infile> [<outfile>]
```

This utility has the following options:

`-d|--dump-fields`

These options dumps (display) all fields in the input file *infile* as they are read.

`-q|--quiet`

These options suppresses informative messages.

`--single-section`

This option checks that exactly one of each section type is present in the input file *infile*. If more than one section of a type is present, an error is displayed.

`--aif`

This option generates an output file *outfile* in nonexecutable AIF output (ARM only, dep recated).

`--bin`

This option generates an output file *outfile* in raw binary format.

`--sxf`

This option generates an output file *outfile* in nShield SEE Executable Format (SXF).

`-n|--no-output`

These options check the input file *infile* without generating any output.

To view the loadable sections of an ELF file, use the following command:

```
elftool --dump-fields <infile>
```

This command displays details of the sections of the file under one of the following categories:

`Read Only`

This category includes program code and constant data (either `Read` or `Read+Execute` permissions).

`Read/Write`

This category includes non-constant data initialized to particular values (`Read+Write` per missions).

`Zero Init`

This category includes non-constant data initialized to zero.

To generate an AIF or SXF format output file correctly, the ELF input file must have the following characteristics:

- The address range for one category of data (for example, `Read Only`) must not overlap with the address range for another (for example, `Read/Write`).
- All `Zero Init` data must come after all `Read/Write` data in memory (that is, `Zero Init` data must occupy a higher memory address).

The default options for most linkers ensure that ELF files meet these requirements.

To convert a ELF file into SXF, a format specifically for SEE machines, use the following command:

```
elftool --sxf <infile> <outfile>
```

SXF format files can be loaded on all existing SEE-enabled HSMs. This is the preferred format.

To convert a ELF file into binary format, use the following command:

```
elftool --bin <infile> <outfile>
```

The output file consists of the `Read Only` data immediately followed by the `Read/Write` data, without a header. This may be useful in applications other than SEE Machine images.

7.3. loadmache

The `loadmache` command-line utility supplied with the Secure Execution Engine (SEE) loads an SEE machine into an SEE-enabled HSM. The hardserver can automatically use this utility to load an SEE machines.

To use this command, you must be logged in to the host computer as a user in the group `nfast` (**Linux**) or as a user who is permitted to create privileged connections (**Windows**).

SEE machines that require support from a host-side `see-*-serv` utility

If your SEE machine requires support from a host-side `see-*-serv` utility, you must run one of those utilities as appropriate to serve the SEE machine before its networking or `stdioe` provisions can work.

7.3.1. Usage

```
loadmache [-m|--module=<MODULE>] [-s|--slot=<SLOT>] [-U|--unencrypted] [-e|--encryptionkey=<IDENT>] [-a|--
sighash=<HASH>] [-n|--noprompt] <machine-filename>
```

In this command, the *machine-filename* parameter specifies the path of the SEE machine. If *machine-filename* is not specified, `loadmache` tries to select a machine from the location specified by the `NFAST_SEE_MACHINEIMAGE_`* environment variables. See Environment variables for more information about environment variables.

7.3.1.1. HSM options

`-m|--module=<MODULE>`

These options specify the hardware security module to use.

`-s|--slot=<SLOT>`

These options specify the slot from which to load cards.

7.3.1.2. SEE machine loading options

`-a|--sighash=<HASH>`

These options specify that the SEE machine is to be signed with a key whose hash is *HASH*.

`-n|--noprompt`

These options specify that you are never prompted for missing smart cards.

`-U|--unencrypted`

These options specify that the SEE machine is to be unencrypted. This is the default. If set, these options override any previously specified `NFAST_SEE_MACHINEENCKEY_`* variable. See Environment variables for more information about environment variables.

-e|--encryptionkey=<IDENT>

These options specify that the SEE machine is to be encrypted with the key identified by *IDENT*. If set, these options override the -U|--unencrypted options.

If neither the -e|--encryptionkey nor the -U|--unencrypted options are specified, a decryption key is used only if the name of a suitable one is found in the location specified by the NFAST_SEE_MACHINEENCKEY_DEFAULT environment variable. See Environment variables for more information about environment variables.

7.4. loadsee-setup

The loadsee-setup command-line utility helps you set up, display, or remove settings in the hardserver configuration file that control the automatic loading of SEE machines.

You can use loadsee-setup for one of three types of action by specifying the appropriate option:

- Specifying the --setup option selects the setup action, used to add a new configuration or replace an existing configuration
- Specifying the --remove option selects the remove action, used to remove an existing configuration (without replacing it)
- Specifying the --display option selects the display action, used to display the configuration of one or all HSMs

7.4.1. Usage

```
loadsee-setup -s|--setup -m <MODULE>
loadsee-setup -r|--remove -m <MODULE>
loadsee-setup -d|--display [-m <MODULE>]
```

7.4.1.1. Action selection

-s|--setup

This option selects the setup action, enabling you to set up the hardware configuration file for the HSM specified by -m|--module=<MODULE> to provide automatic loading for the SEE machine specified by -M|--machine=<MACHINE>.sar.

You must always specify the -m|--module=<MODULE> and -M|--machine=<MACHINE>.sar options when using the --setup option. See

> the comments in the hardserver configuration file for information about the effects of specifying or omitting other options.

`-r|--remove`

This option selects the `remove` action, enabling you to remove settings that control auto matic SEE machine loading from the hardware configuration file for the HSM specified by `-m|--module=<MODULE>`

`-d|--display`

This option selects the `display` action, enabling you to display the current configuration of automatic SEE machine loading for the HSM specified by `-m|--module=<MODULE>` or, if no HSM is specified, all HSMs in the Security World.

7.4.1.2. Setup options

`-M|--machine=<MACHINE>.sar`

This option specifies the SEE machine file (packed as a SAR). You must supply a value for this option when using setup mode.

`-U|--userdata=<USERDATA>.sar`

This option specifies the name of the `userdata` file (packed as a SAR) to be passed to SEE machine.

`-k|--key=<IDENT>`

This option identifies the `seeconf` key protecting the SEE machine. You must supply this option is the SEE machine is encrypted. Only HSM-protected keys are supported.

`-S|--sighash=<HASH>`

This option identifies the hash of the key that the SEE machine is signed with. You only need to supply this option if the SEE machine is encrypted and you are using a dynamic SEE feature. This option is not required if the SEE machine is not encrypted or if you have the `GeneralSEE` static feature.

`-p|--published-object=<NAME>`

This option specifies the `PublishedObject` name to use for publishing the `KeyID` of the started SEE machine.

`-P|--postload-prog=<PROGRAM>`

This option specifies the post-load program to be run after the SEE machine is loaded.

In most cases, SEE machines using the `bsdlib`/`glibc` architecture

> should supply the value `bsdlib`/`glibc` to specify use of the provided `postload-bsdlib` program.

`-A|--postload-args="<ARGUMENTS>"`

This option specifies an argument string to pass to the post-load program specified by the `--postload-prog` option. Supply the individual arguments within the double quotation marks, each argument separated from the next by a single space.

7.4.1.3. General options

`-m|--module=<MODULE>`

This option specifies the HSM with the hardware configuration file that is to be acted upon by the command. You must supply a value for this option in either setup or remove mode.

`-c|--configfile=<FILENAME>`

This option specifies name of (or path to) the hardserver configuration file to be acted upon by the command. The default is `/opt/nfast/kmdata/config/config` (**Linux**) or `%NFAST_KMDATA%\config\config` (**Windows**).

`-f|--force`

Setting this option allows the command to make configuration changes without prompting you.

`--no-reset`

This option prevents resetting HSMs with changed configurations.

7.4.2. Output

7.4.2.1. loadsee-setup --setup

This section provides an example of `loadsee-setup` used in `--setup` mode.

When `--setup` mode is specified, the only other required options are `-m|--module=<MODULE>` and `-M|--machine=<MACHINE>.sar`. However, if you supply the `-A|--postload-args="<ARGUMENTS>"` option, you must also supply the `-P|--postload-prog=<PROGRAM>` option.

To set up a hardware configuration file to provide automatic loading for an SEE machine, run a command similar to the following Solo XC example:

```
loadsee-setup --setup -m1 --machine /tmp/test.sar --postload-prog=glibc --postload-args="--provision stdoe
```

```
--userdata-sar /tmp/userdata.sar --trace"
```

If automatic SEE machine loading has already been configured for the specified HSM, `loadsee-setup` warns you before it is overwritten:

```
Module #1 new SEE configuration saved, new configuration follows:
Module #1:
  Machine file:                /tmp/test-helloworld.sar
  Userdata file:
  WorldID published object:
  Postload helper:             glibc
  Postload args:               --provision stdoe --userdata-sar /tmp/test.cpio.sar
--trace
Clear modules now to reload new configuration? (yes/no): yes
```

You can use the `-f|--force` option to bypass this warning and overwrite the existing configuration.

After setting up the configuration, `loadsee-setup` resets the affected HSM (unless you specified the `--no-reset` option).

7.4.2.2. loadsee-setup --remove

This section provides an example of `loadsee-setup` used in `--remove` mode.

When `--remove` mode is specified, the only other required option is `-m|--module=<MODULE>`. This specifies the HSM with the hardserver configuration file that needs the settings for automatic SEE machine loading removed.

To remove settings for automatic SEE machine loading from an HSM's hardware configuration file, run a command similar to the following example:

```
loadsee-setup --remove -m1
```

If the HSM specified by `-m|--module=<MODULE>` does not exist or is not currently configured to automatic SEE machine loading configured, an error is displayed. Otherwise, the current configuration is displayed and `loadsee-setup` prompts you to continue:

```
Module #1:
  Machine file:                /tmp/test-helloworld.sar
  Userdata file:
  WorldID published object:
  Postload helper:             glibc
  Postload args:               --provision stdoe --userdata-sar /tmp/test.cpio.sar
--trace
Erase this configuration? (yes/no): yes
Module #1 SEE auto-loading configuration removed.
Clear modules now to reload new configuration? (yes/no): yes
```

You can use the `-f|--force` option to bypass warnings and remove the existing configuration without being prompted.

After removing the configuration, `loadsee-setup` resets any HSM with a configuration that has changed (unless you specified the `--no-reset` option). After running `loadsee-setup` command in `--remove` mode, no SEE machines are automatically loaded onto the specified HSM.

7.4.3. loadsee-setup --display

This section provides an example of `loadsee-setup` used in `--display` mode.

You are not required to specify any additional options with `--remove` mode. You can specify the `-m|--module=<MODULE>` option to display the settings for automatic SEE machine loading in a particular HSM's hardserver configuration file; without specifying this option, `loadsee-setup` displays the settings for automatic SEE machine loading in the hardserver configuration files for any HSM in the Security World for which these settings exist.

To display settings for automatic SEE machine loading for all HSMs, run a command similar to the following example:

```
$ loadsee-setup --display
```

This command produces output similar to the following:

```
Module #1:
  Machine file:                /tmp/test-helloworld.sar
  Userdata file:
  WorldID published object:
  Postload helper:             glibc
  Postload args:               --provision stdoe --userdata-sar /tmp/test
```

7.5. hsc_loadseemachine

The `hsc_loadseemachine` utility enables you to publish an SEE machine. The utility:

1. Loads an SEE machine into each HSM configured.
2. Publishes a newly created SEE world, if appropriate.

7.5.1. Usage

```
hsc_loadseemachine [<options>]
```

7.5.1.1. Options

`-m|--module`

This option specifies the HSM number into which the configuration data must be read. The default value is `0`.

The SEE machine can be loaded only if you specify this option. If you do not specify this option, the utility examines the configuration file to check the changes that are made to the `load_seemachine` section and then reset any HSM that has had its entry modified. The hardserver loading script then calls `hsc_loadseemachine -m MODULE` for each HSM that has been reset.

`-c|--configfile=<FILENAME>`

This option specifies the name of the configuration file that must be read.

7.6. nfkmverify

The `nfkmverify` command-line utility verifies key generation certificates. You can use `nfkmverify` to confirm how a particular Security World and key are protected. It also returns some information about the Security World and key.

The `nfkmverify` utility compares the details in the ACL of the key and those of the card set that currently protects the key.

A key that has been recovered to a different card set shows a discrepancy for every respect that the new card set differs from the old one. For example, a key recovered from a 2-of-1 card set to a 1-of-1 card set has a different card-set hash and a different number of cards, so two discrepancies are reported. The discrepancy is between the card set mentioned in the ACL of the key and the card set by which the key is currently protected (that is, the card set mentioned in the key blobs).

A key that has been transferred from another Security World shows discrepancies and fails to be verified. We recommend that you verify keys in their original Security World at their time of generation.

If you must replace your Security World or card set, we recommend that you generate new keys whenever possible. If you must transfer a key, perform key verification immediately before transferring the key; it is not always possible to verify a key after transferring it to a new Security World or changing the card set that protects it.

7.6.1. Usage

```
nfkmverify [-f|--force] [-v|--verbose] [-U|--unverifiable] [-m|--module=<MODULE>] [appname ident [appname ident
[...]]]
```

7.6.1.1. Help options

-h|--help

This option displays help for `nfkmverify`.

-V|--version

This option displays the version number for `nfkmverify`.

-u|--usage

This option displays a brief usage summary for `nfkmverify`.

7.6.1.2. Program options

-m|--module=<MODULE>

This option performs checks with module *MODULE*.

-f|--force

This option forces display of an output report that might be wrong.

-U|--unverifiable

This option permits operations to proceed even if the Security World is unverifiable.

If you need the `-U|--unverifiable` option, there may be some serious problems with your Security World.

-v|--verbose

This option prints full public keys and generation parameters.

-C|--certificate

This option checks the original ACL for the key using the key generation certificate. This is the default.

-L|--loaded

These options check the ACL of a loaded key instead of the generation certificate.

-R|--recov

This option checks the ACL of the key loaded from the recovery blob.

`--allow-dh-unknown-sg-group`

This option allows an operation to proceed even if a Diffie-Hellman key is using an unrecognized Sophie-Germain group.

7.6.2. Output

Output returned from `nfkmverify` can take a variety of forms, depending on the parameters of the given key generation certificate, Security World, and key concerned. Examples of pos sible output resulting from several different situations are provided below.

Under normal circumstances, issuing a command of the form:

```
nfkmverify --verbose --unverifiable myapp o20010621a13h25m02
```

returns output of the form:

```
** [Security world] **
   1 Administrator Cards
        (Currently in Module #1 Slot #0: Card #1)
       Cardset recovery ENABLED
       Passphrase recovery disabled
       Strict FIPS 140 level 3 (does not improve security) disabled
       SEE application nonvolatile storage disabled
       real time clock setting disabled
       SEE debugging disabled
       Generating module ESN 0A42-E645-7A75 currently #1 (in same incarnation)
** [Application key myapp o20010621a13h25m02] **
    [Named 'test Thu, 21 Jun 2001 13:25:02 +0100']
        Useable by HOST applications.
        Recovery ENABLED.
        MODULE-ONLY protection
        Type RSAPrivate 1024 bits keygenparams.type= RSAPrivate 2
               .params.rsaprivate.flags= none 0x00000000
                                                    .lenbits= 0x00000400 1024
                                                    .given_e absent
                                                    .nchecks absent

    Generating module ESN 0A42-E645-7A75 currently #1 (in same incarnation)
        nCore hash 23a901f3329aa9e29cd79d3bb7b32d549b725fc3
        public_half.type= RSAPublic 1
                  .data.rsapublic.e= 4 bytes
                                                    00010001

                                                    .n= 128 bytes
        8a6ab219 183de558 48c8379e 840895ff 0ba64bae 392848c6 c0aeb7f9 d10b046d
        4a214b70 4878b518 8e599c69 1cd61db0 bab4f852 425c70f5 b9c000e5 4ceda15f
        c062b5dd 01852380 f70275a1 870a6947 68ef59f0 db5d2e84 d6ae8dc1 7542e94d
        adedece8 cb3c9fb6 98fab8af 52c94137 a76ab7dd 38648134 0df55ca8 2f45e8b7

Verification successful, check details above.
```

Output of the form shown above indicates successful verification of the relevant key gener ation certificate.

The following examples indicate forms of output that could be returned if you try to verify the generation certificate of a key generated in a Security World that was created with an insufficiently up-to-date version of Security World for nShield.

In such a case, issuing a command of the form:

```
nfkmverify --verbose myapp spong
```

returns output of the form:

```
PROBLEM: no world generation certificates
PROBLEM: application key myapp spong: no key generation signature
 2 issues found, NOT VERIFIED
```

Adding the `--unverifiable` option to the same command:

```
nfkmverify --verbose --unverifiable myapp spong
```

returns output of the form:

```
PROBLEM: application key myapp spong: no key generation signature
1 issues found, NOT VERIFIED
```

Then, also adding the `--force` option to this same command:

```
nfkmverify --force --verbose --unverifiable myapp spong
```

returns output of the form:

```
PROBLEM: application key myapp spong: no key generation signature
PROBLEMS BUT FORCING POSSIBLY-WRONG OUTPUT
 ** [Security world] **
     UNVERIFIED SECURITY WORLD !
          proceeding anyway as requested
 ** [Application key myapp spong] **
    [Not named]
         Useable by HOST applications.
         Recovery ENABLED.
         MODULE-ONLY protection

 1 issues found, NOT VERIFIED
```

8. Environment variables

This appendix describes the environmental variables used by Security World Software to control SEE functionality:

Variable	Description
NFAST_SEE_MACHINEENCKEY_DEFAULT	This variable is the name of the SEEConf key needed to decrypt SEE-machine images. Running the command loadmache --encryptionkey=<IDENT> (or loadmache --unencrypted) overrides any value set by this variable.
NFAST_SEE_MACHINEENCKEY_<module>	This variable is the name of the SEEConf key needed to decrypt the SEE-machine image targeted for the specified HSM. It overrides NFAST_SEE_MACHINEENCKEY_DEFAULT for the specified HSM. Running the command loadmache --encryptionkey=<IDENT> (or loadmache --unencrypted) overrides any value set by this variable.
NFAST_SEE_MACHINEIMAGE_DEFAULT	This variable is the path of the SEE machine image to load on to any HSM for which a specific image is not defined. Supplying the *machine-filename* parameter when running the loadmache command-line utility overrides this variable. This variable is not affected when running the loadsee-setup or hsc_loadseemachine utilities.
NFAST_SEE_MACHINEIMAGE_<module>	This variable is the path of the SEE machine image to load on to the specified HSM. If set, this variable overrides the use of NFAST_SEE_MACHINEIMAGE_DEFAULT for the specified HSM. Supplying the *machine-filename* parameter when running the loadmache command-line utility overrides the NFAST_SEE_MACHINEIMAGE_<module> variable. This variable is not affected when running the loadsee-setup or hsc_loadseemachine utilities.
NFAST_SEE_MACHINESIGHASH_DEFAULT	This variable is the default key hash of the vendor signing key (seeinteg) that signs SEE machine images. This variable is only required if you are using a dynamic SEE feature with an encrypted SEE machine. Running the command loadmache --sighash=<HASH> any value set in this variable.

Variable	Description
NFAST_SEE_MACHINESIGHASH_<module>	This variable is the key hash of the vendor signing key (seeinteg) that signs SEE machine images for the specified HSM. It overrides NFAST_SEE_MACHINESIGHASH_DEFAULT for the specified HSM. This variable is only required if you are using a dynamic SEE feature with an encrypted SEE machine. Running the command loadmache --sighash=<HASH> any value set in this variable.

Windows-only

When the hardserver is running normally as a service, these are System variables only; not the User Variables. The hardserver checks first for these variables, but if any given value is not set in the environment, the hardserver next searches for a string value in the Registry under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\nFast Server\Environment.

For information on additional (non-SEE) environment variables used by Security World Software, see Environment variables.

9. SEElib functions

The file `seelib.h` contains wrapper functions for the software interrupts.

9.1. SEElib_init

```
extern void SEElib_init(void);
```

This function initializes the `SEElib` library.

It also checks that the SWI interface that was implemented by the nShield core matches the version that the SEE machine implements.

This function does not return on error.

9.2. SEElib_RecProcessThreads

```
int
SEElib_RecProcessThreads(void);
```

This function returns the recommended number of processing threads on this system.

9.3. SEElib_StartProcessorThreads

```
struct ProcessThreadCtx; /* User-defined */
typedef struct SEElib_ProcessContext
{
  struct ProcessThreadCtx *uc;

  unsigned char *iobuf;
  int iobuf_maxlen;
}
  SEElib_ProcessContext;

typedef struct ProcessThreadCtx * (*SEEJobInitFn) (SEElib_ProcessContext *pC);
/* Function called during thread initialisation */
typedef int (*SEEJobFn) ( SEElib_ProcessContext *pC, M_Word tag, int in_len );
/* Function to process an SEEJob; data is sent in & out via pC->iobuf.
Returns length being returned.
*/
extern int SEElib_StartProcessorThreads(int nthreads, int stacksize, SEEJobInitFn
pfnInit, SEEJobFn pfnProcess);
```

This function causes the SEE library to start a number of processing threads. Each thread has its own `SEElib_ProcessContext` allocated, which remains constant throughout the life

of the thread.

A working buffer for a given thread is allocated; the `iobuf` member points to this buffer and `iobuf_maxlen` is set to the size. Data for the `SEEJob` is passed in and out through this buffer.

For each thread, the supplied `SEEJobInitFn` is called first, and the `ProcessThreadCtx` pointer it returns is stored in the `SEElib_ProcessContext` structure. This structure is typically some convenient thread-local storage. The pointer may be NULL if it is not required.

When a job arrives for the given thread, the supplied `SEEJobFn` is called. It is passed the `SEElib_ProcessContext` pointer `pC`, a tag, and a length (`in_len`). The `SEEJob` data is at `pC→iobuf`, length `in_len`. The tag is merely for information. The function should process the data and leave a reply at `pC→iobuf`. The return value from the function indicates the number of bytes to be returned from this buffer.

9.4. SEElib_GetUserDataLen

```
extern M_Word SEElib_GetUserDataLen (void);
```

This function gets the length in bytes of the `UserData` block that was passed in to create this SEE World. The function returns 0 if the UserData block has been freed with `SEElib_ReleaseUserData()`.

9.5. SEElib_ReadUserData

```
extern int SEElib_ReadUserData ( M_Word offset, unsigned char *buf, M_Word len );
```

This function reads selected bytes from the `UserData` block, starting at *offset* bytes in and continuing for *len* bytes. It returns an `M_Status` value.

9.6. SEElib_ReleaseUserData

```
extern void SEElib_ReleaseUserData(void);
```

This function frees the resources associated with the `UserData` block. Typically, if an SEE machine copies the `UserData` block into some internal format on initialization, it should call this function on completion to avoid having two copies of the data in memory.

9.7. SEElib_InitComplete

```
extern void SEElib_InitComplete( M_Word status );
```

This function must be called as soon as the SEE World has been initialized. This call must be made as soon as the SEE World is ready to accept jobs or has decided that it cannot accept jobs.

The `status` value forms the `initstatus` value in the reply to the `CreateSEEWorld` nCore API command.

9.8. SEElib_AwaitJob

```
extern int SEElib_AwaitJob( M_Word *tag_out, , unsigned char *buf, M_Word *len_io );
```

This function blocks and waits for the next `SEEJob` in from the nShield core. On entry, `*buf` and `*len_io` give the base and length of a buffer area to receive the job. On return, `*len_io` is set to the length delivered (if the job is received successfully). This buffer is a copy of the `seeargs` field that was sent in to the `SEEJob` command.

The `*tag_out` value is the tag for this command. It must be returned in the `SEElib_Return-Job` so that the nShield core associates the reply with this command.

The `SEElib_AwaitJob` function returns an `M_Status`, which is only likely to be `OK` or `Buffer-Full`.

If you use `SEElib_StartProcessorThreads()`, it calls this function automatically, and you should not call this function yourself.

9.9. SEElib_StartTransactListener

```
extern void SEElib_StartTransactListener(void);
```

This function starts the thread that listens for `SEElib_Transact` calls and dispatches them. This function must be called before any use is made of `SEElib_Transact`.

9.10. SEElib_Transact

```
extern int SEElib_Transact(struct M_Command *cmd, struct M_Reply *buf);
```

This function marshals a command, submits it, waits for the response, and unmarshals it into a reply structure.

9.11. SEElib_MarshalSendCommand

```
extern int SEElib_MarshalSendCommand(M_Command *cmd);
```

This function marshals a command and places it on the input queue for processing by the nShield core.

The command takes a reference to an `M_Command` structure, as described in the *nCore Code-Safe API Documentation*.

The SEE machine can submit any of the nCore API commands listed in the Basic commands and Key-Management commands sections of the *nCore CodeSafe API Documentation* except:

- `RetryFailedModule`
- `GetWhichModule`
- `MergeKeyIDs`.

If the SEE machine attempts to submit one of these commands, the nShield core returns a response with the status code `NotAvailable`.

The `SEElib_MarshalSendCommand` function returns an `M_Status` value. This value is `OK` if the command was marshalled and transferred to the nShield core correctly.

Do not mix calls to `SEE_Transact()` and `SEElib_MarshalSendCommand()` and `SEElib_GetUnmarshalResponse()`, because the replies may be misdirected.

9.12. SEElib_GetUnmarshalResponse

```
extern int SEElib_GetUnmarshalResponse(M_Reply *buf);
```

If there is a reply in the input queue for this SEE World, this function returns the first job in the queue. Otherwise, it blocks and waits for the nShield core to return a job.

On return, `M_Reply` contains the unmarshalled reply.

The `SEElib_GetUnmarshalResponse` function returns an `M_Status` value. This value is `OK` if the

reply was unmarshalled successfully. The return of this value does not necessarily mean that the command was completed successfully, only that the reply was unmarshalled. You must also check the `M_Status` within the reply.

9.13. SEElib_FreeCommand

```
extern int SEElib_FreeCommand(struct M_Command *cmd);
```

This function frees a command structure and is equivalent to the generic stub function `NFastApp_FreeCommand` (described in the *nCore CodeSafe API Documentation*).

9.14. SEElib_FreeReply

```
extern int
SEElib_FreeReply(struct M_Reply *reply);
```

This function frees a reply structure and is equivalent to the generic stub function `NFastApp_FreeReply` (described in the *nCore CodeSafe API Documentation*).

9.15. SEElib_ReturnJob

```
extern void SEElib_ReturnJob( M_Word tag, const unsigned char *data, unsigned int len );
```

This function returns an `SEEJob` reply to the nShield core so that the core can pass it to the calling application.

If you use the `SEElib_StartProcessorThreads()` function, it calls `SEElib_ReturnJob()` for you.

The tag field must match the tag supplied in the `SEElib_AwaitJob()` call that created the job.

The given data is copied away and forms the `seereply` field of the `SEEJob` reply (see the description of the `SEEJob` command in the *nCore CodeSafe API Documentation*).

9.16. SEElib_SubmitCoreJob

```
extern int SEElib_SubmitCoreJob( const unsigned char *data, unsigned int len );
```

This function puts a job on the input queue for processing by the core. The byte block is passed in `data` and `len`. It should be a full marshalled `M_Command` with a valid tag at the start.

This function returns an `M_Status`, which is typically `OK` or `BufferFull` (if `len` is too big).

9.17. SEElib_GetCoreJob

```
extern int SEElib_GetCoreJob ( unsigned char *buf, M_Word *len_io );
```

This function blocks and waits for a job submitted to the core to be returned. On entry, `buf` points to a buffer of length `(*len_io) max`. On exit, if successful, `*len_io` is the length of bytes returned.

This function returns an `M_Status`, which is typically `OK` or `BufferFull` (if `len_io` is too big).

9.18. SEElib_GetUserDataLen

```
extern M_Word SEElib_GetUserDataLen ( void );
```

This function gets the length in bytes of the UserData block passed in to create this SEE World.

If this data has been discarded because `SEElib_ReleaseUserData()` has been called, this function returns `0`.

9.19. SEElib_Submit

```
extern int SEElib_Submit(M_Command *cmd, M_Reply *reply, PEVENT ev, SEElib_ContextHandle tctx);
```

This function submits the command specified in `cmd`. The transaction listener thread calls EventSet `ev`, if `ev` is non-NULL, when the reply returns for this command. The reply is unmar shalled into `reply` and `tctx` is returned to the caller in `SEElib_Query`.

Unlike `SEElib_SubmitCoreJob` this function can be called at the same time as another thread is blocking in `SEElib_Transact`.

`SEElib_StartTransactListener` must have been called before this function is called.

9.20. SEElib_Query

```
extern int SEElib_Query(M_Reply **replyp, SEElib_ContextHandle *tctx_r);
```

This function is called to receive a reply that is being held by the transaction listener thread. It is typically called after having been woken from `EventWait` as a result of the transaction lis tener thread posting to the event passed in to `SEElib_Submit`.

If `*replyp` is NULL, `SEElib_Query` accepts any returned reply, and `*replyp` is changed to point to that reply. If `*replyp` is not NULL, the function accepts the reply specified; other replies are queued internally.

`tctx_r` may be NULL. If it is not, the `tctx` used when submitting the reply is stored in `*tctx_r`. `SEElib_Query` can return, in addition to the usual return values, `TransactionNotYet-Complete` if the reply (or any reply if `*replyp` was NULL) has not come back from the core yet.

`SEElib_StartTransactListener` must have been called before this function is called.

9.21. SEElib_StartSEEJobListener

```
extern int SEElib_StartSEEJobListener(PEVENT ev);
```

This function starts the `SEEJob` listener thread which blocks calling `SEElib_AwaitJob`, caches the new job and then sets the event `ev` if `ev` is non-NULL.

Use `SEElib_QuerySEEJob` to receive any `SEEJobs` that have been cached by this listener thread, followed by `SEElib_ReturnJob` to reply to the `SEEJob`, then followed by `SEElib_Re-leaseSEEJob` to free the buffer.

It is safe to call this function multiple times. Calls after the first call will have no effect.

9.22. SEElib_QuerySEEJob

```
extern M_Status SEElib_QuerySEEJob( M_Word *tag_out, unsigned char **buf, M_Word *len );
```

This function is called to receive a `SEEJob` that is being held by the `SEEJob` listener thread. It is typically called after having been woken from `EventWait` as a result of the `SEEJob` listener thread setting the event passed in to `SEElib_StartSEEJobListener`.

`buf` is set to the buffer containing the SEEJob, `len` is set to the length of the data contained in `buf`.

This function returns `TransactionNotYetComplete` if there were no outstanding `SEEJobs`.

9.23. SEElib_ReleaseSEEJob

```
extern void SEElib_ReleaseSEEJob( unsigned char **buf );
```

This function is called to release a buffer which was returned from `SEElib_QuerySEEJob`. This function must be called after the buffer specified by `buf` in a call to `SEElib_QuerySEEJob` has been finished with. This function is safe to call even if `*buf` is NULL. In addition, this function sets `*buf` to NULL on completion.

10. Differences between glibc and bsdlib (SoloXC only)

In order to provide CodeSafe developers the ability to write standard POSIX calls and be able to run in the SEE environments, gcc wrappers are used in Solo XC programs to override certain standard GNU C library (`glibc`) functions. Older SEE programs built to run on Solo+ use the BSD C Library (`bsdlib`). For example, both CodeSafe and Libc, have a definition for the function socket:

```
socket(int __domain, int __type, int __protocol)
```

At link time, the function call is overridden and resolved to the CodeSafe implementation. A linker options is used to accomplish that.

```
-Wl,-wrap=socket
```

The standard POSIX `socket()` function can still be used calling `real_socket()`. The applicability of the standard (`real_*`) familiarity of functions is limited in the SEE environment due to embedded system constraints.

All the wrapped functions were replaced by equivalent ones with the underlying IPC support to communicate with nShield core and provide the same functionality as in legacy systems.

List functions that were wrapped and redefined:

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `read()`
- `write()`
- `send()`
- `setsockopt()`
- `poll()`
- `select()`

10.1. glibc Compatibility exceptions

As a consequence of some function redefinitions and the underlying differences, some stan dard C functions may not work as expected in CodeSafe.

`FILE *fdopen( int fd, const char *mode)`: associates a stream with an existing file descriptor, `fd`. In the case of a socket `fd` (returned by CodeSafe socket() implementation) the association result may fail or cause unexpected errors in subsequent calls. Developers should avoid using `fdopen` with non-standard Unix file descriptors.

11. Allowlist for SEE machines

Classic and GLIBC SEE machines are restricted to a subset of Linux system calls they can execute.

An SEE machine that attempts to execute a system call that is not allowed will be immediately terminated by a safeguarding process.

Allowed system calls	
1 __NR_exit	2 __NR_fork
3 __NR_read	4 __NR_write
5 __NR_open	6 __NR_close
7 __NR_waitpid	8 __NR_creat
9 __NR_link	10 __NR_unlink
11 __NR_execve	12 __NR_chdir
13 __NR_time	15 __NR_chmod
19 __NR_lseek	21 __NR_mount
22 __NR_umount	24 __NR_getuid
29 __NR_pause	33 __NR_access
37 __NR_kill	38 __NR_rename
39 __NR_mkdir	40 __NR_rmdir
41 __NR_dup	42 __NR_pipe
45 __NR_brk	49 __NR_geteuid
54 __NR_ioctl	60 __NR_umask
63 __NR_dup2	64 __NR_getppid
65 __NR_getpgrp	78 __NR_gettimeofday
83 __NR_symlink	85 __NR_readlink
90 __NR_mmap	91 __NR_munmap
94 __NR_fchmod	99 __NR_statfs
102 __NR_socketcall	106 __NR_stat
107 __NR_lstat	108 __NR_fstat
114 __NR_wait4	119 __NR_sigreturn

Allowed system calls	
120 __NR_clone	125 __NR_mprotect
140 __NR_llseek	141 __NR_getdents
145 __NR_readv	146 __NR_writev
160 __NR_sched_get_priority_min	162 __NR_nanosleep
163 __NR_mremap	172 __NR_rt_sigreturn
173 __NR_rt_sigaction	174 __NR_rt_sigprocmask
175 __NR_rt_sigpending	176 __NR_rt_sigtimedwait
177 __NR_rt_sigqueueinfo	178 __NR_rt_sigsuspend
179 __NR_pread64	181 __NR_chown
182 __NR_getcwd	190 __NR_ugetrlimit
195 __NR_stat64	196 __NR_lstat64
197 __NR_fstat64	202 __NR_getdents64
204 __NR_fcntl64	205 __NR_madvise
207 __NR_gettid	221 __NR_futex
232 __NR_set_tid_address	234 __NR_exit_group
250 __NR_tgkill	252 __NR_statfs64
286 __NR_openat	300 __NR_set_robust_list
326 __NR_socket	327 __NR_bind
328 __NR_connect	329 __NR_listen
330 __NR_accept	331 __NR_getsockname
332 __NR_getpeername	333 __NR_socketpair
334 __NR_send	335 __NR_sendto
336 __NR_recv	337 __NR_recvfrom
338 __NR_shutdown	339 __NR_setsockopt
340 __NR_getsockopt	