nShield Security World

CodeSafe 5 v13.6.5 Developer Guide

08 January 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, including 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 are 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, timestamp servers, audit loggers, digital signature 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 protected keys. The code itself is signed to provide additional protection.

2. Overview of CodeSafe 5

2.1. Applications as container images

In CodeSafe 5, the application is a container image, meaning a complete filesystem image that can contain multiple executables, libraries, scripts, and data files.

This has the following benefits:

- Data files can be written to the local filesystem and persisted over container shutdown and restart.
- The application can comprise multiple co-operating processes. This can enhance security by separating memory spaces and reliability by allowing individual processes to be restarted if they crash or leak memory.
- Third-party or pre-existing Linux source code can be built and run without modification.
- Standalone tools can be executed as subprocesses.
- Dynamically-loaded libraries work in a regular way. Code architectures that make use of plug-in modules make code development easier and reduce the attack surface by excluding unwanted code.

2.2. Easy and fast network connectivity

nShield 5 HSMs and CodeSafe 5 containers are logically connected via TCP/IP networking. The container running the SEE Machine can receive incoming connections from the host side app, establishing two-way communication between host side app and SEE machine. Existing software that makes use of incoming or outgoing network connections can run with little or no modifications.

Kernel-implemented networking provides good performance both for throughput and for latency.

2.3. 'Secure by default' client communication

The CodeSafe 5 execution environment includes both a configurable firewall and an SSH server. The firewall is set according to configuration in the signed CodeSafe 5 application package so that only the network ports required by the application are allowed. The SSH server allows a secure tunnel to be established to the CodeSafe 5 application. The client credentials required to access this tunnel can be configured using the support tools.

This means that applications, including applications ported from older CodeSafe SEE machines, can benefit from strong authentication of their clients and protection from unauthorized network traffic without additional code.

2.4. Better language support

The CodeSafe 5 SDK supports:

- C and C++
- Python 3.8

  The `nfpython` module provides easy access to nCore API commands.

  The container environment has a regular Linux filesystem and supports system calls for network and file I/O, so a wide range of standard and third-party Python modules can be used without modification.

CodeSafe applications can be written using mixed languages with the usual range of IPC and calling mechanisms available to the developer.

2.5. Developer authentication

CodeSafe 5 uses Entrust X.509 certificates to link the CodeSafe application to a real-world developer identity through code signing.

This allows the administrator of an HSM to, for example, restrict the HSM to authorized in-house applications or to those provided by trusted development partners.

3. Install the CodeSafe 5 SDK on Linux

1. Make sure that the following nShield ISO images are available locally:
   - `SecWorld_Lin64-13.x.y.iso`
   - `Codesafe_Lin64-13.x.y.iso`

   Where `<x.y>` are the same versions for Security World and CodeSafe.

2. Create a mount directory for each ISO:

```
mkdir ~/secworld_iso_mountpoint
mkdir ~/codesafe_iso_mountpoint
```

3. Mount the ISO images to their respective directories:

```
sudo mount <PATH_TO>/SecWorld_Lin64-13.x.y.iso ~/secworld_iso_mountpoint/
sudo mount <PATH_TO>/Codesafe_Lin64-13.x.y.iso ~/codesafe_iso_mountpoint/
```

   The nShield CodeSafe 5 hostside is located in tarballs under:

```
ls ~/codesafe_iso_mountpoint/linux/amd64/
csdref.tar.gz  csd.tar.gz
```

   The nShield Security World hostside is located in tarballs under:

```
ls ~/secworld_iso_mountpoint/linux/amd64/
ctd.tar.gz   devref.tar.gz  javasp.tar.gz  ncsnmp.tar.gz
ctls.tar.gz  hwsp.tar.gz    jd.tar.gz      raserv.tar.gz
```

4. Untar the tarballs into the root directory:

```
tar -zxvf ~/codesafe_iso_mountpoint/linux/amd64/csd.tar.gz -C /
tar -zxvf ~/codesafe_iso_mountpoint/linux/amd64/csdref.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/ctd.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/devref.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/javasp.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/ncsnmp.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/ctls.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/hwsp.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/jd.tar.gz -C /
tar -zxvf ~/secworld_iso_mountpoint/linux/amd64/raserv.tar.gz -C /
```

   This installs the nShield CodeSafe 5 SDK to `/opt/nfast/c/csd5` and the nShield CodeSafe 5 SDK Python files to `/opt/nfast/python3/csd5`.

4. Install the CodeSafe 5 SDK on Windows

4.1. Prerequisites

Make sure that the following nShield ISO images are available locally:

- `SecWorld_Windows-13.x.y.iso`
- `Codesafe_Windows-13.x.y.iso`

Where `<x.y>` are the same versions for Security World and CodeSafe.

4.2. Install the Security World Software

1. Log in as Administrator or as a user with local administrator rights.
2. Mount the Security World Software ISO image and navigate into the mounted directory.
3. Launch `setup.msi`.
4. Follow the on-screen instructions.
5. Accept the license terms and select **Next** to continue.
6. Specify the installation directory and select **Next** to continue.
7. Select **Install**.
8. Select **Finish** to complete the installation.

4.3. Install CodeSafe 5

1. Mount the CodeSafe 5 SDK ISO image and navigate into the mounted directory.
2. Launch `setup.msi`.
3. Follow the on-screen instructions.
4. Accept the license terms and select **Next** to continue.
5. Specify the installation directory and select **Next** to continue.
6. Select **Install**.
7. Select **Finish** to complete the installation.

   This installs the nShield CodeSafe 5 SDK `C:\Program Files\nCipher\nfast\c\csd5` and the nShield CodeSafe 5 SDK Python files to `C:\Program Files\nCipher\nfast\python3\csd5`.

5. nShield 5c Codesafe 5 Configuration

To use CodeSafe 5 with a nShield 5c you must generate and exchange launcher service keys between the client and the nShield 5c. These keys are essential for secure communication and access to the launcher service on the module. See CodeSafe setup for the nShield 5c for more information.

6. Build CodeSafe 5 SDK apps

6.1. General SDK use

The CodeSafe 5 SDK provides the tools necessary to build and run SEE machines on nShield 5 HSMs. The CodeSafe 5 SEE machines are containerized. The SDK provides the structure of the container, including a root file system, libraries required for communication with the nCore API, and libraries to enable communication between the SEE machine and the host. The SDK provides libraries for development, libraries built for maintaining backwards compatibility for legacy applications, a root file system with libraries useful for development of new applications, such as `libglib` and `libc`, and useful binaries including `touch`, `cat`, `grep`.

6.2. Prerequisites

GCC 8.x or later.

6.3. SDK file structure overview

6.3.1. SDK location

The default installation location of the CodeSafe 5 SDK is:

- **Linux**: `/opt/nfast/c/csd5/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\`

Some tools required for SEE machine operations might be found elsewhere in the main install. For example, `csadmin`, which enables loading, starting, and stopping SEE machines, is installed in the following default locations:

- **Linux**: `/opt/nfast/bin/csadmin`
- **Windows**: `C:\Program Files\nCipher\nfast\bin\csadmin` (Windows)

These cases are described in the following sections as required.

6.3.2. Container root file system

The container root file system is located in:

- **Linux**: `/opt/nfast/c/csd5/rootfs/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\rootfs\`

This root file system contains two main parts: binary files and libraries.

6.3.2.1. Binaries

`rootfs/bin/` (Linux) or `rootfs\bin\` (Windows) contains many useful common Linux binaries that you might need within the container such as `cat`, `grep`, and `touch`.

`rootfs/sbin/` (Linux) or `rootfs\sbin\` (Windows) contains the `init` script for the container.

6.3.2.2. Libraries

`rootfs/lib/` and `rootfs/usr/lib/` (Linux) or `rootfs\lib\` and `rootfs\usr\lib\` (Windows) contain various useful libraries a developer might need, such as `libglib` and `libc`. Some of these libraries are also essential for the proper running of the container and execution of various examples.

6.3.3. CMake

The SDK installs a directory which includes CMake toolchains used for building example SEE machines:

- **Linux**: `/opt/nfast/c/csd5/cmake`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\cmake`

These toolchains can serve as examples themselves for creating custom toolchains.

6.3.4. Include directories

The SDK provides two directories with header files that can be included along with their respective libraries to provide additional functionality in SEE machines. These headers are stored in:

- **Linux**:
  - `/opt/nfast/c/csd5/gcc/*`
  - `/opt/nfast/c/csd5/include-see/*`
- **Windows**:
  - `C:\Program Files\nCipher\nfast\c\csd5\gcc\*`

- `C:\Program Files\nCipher\nfast\c\csd5\include-see\*`

6.3.5. SEE specific libraries

The C libraries which are specific to SEE machines, including `seelib.a` and `librtusr.a`, are located in:

- **Linux**: `/opt/nfast/c/csd5/lib-ppc64-linux-musl/\*`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\lib-ppc64-linux-musl\*`

These libraries must be included to enable critical SEE machine functionality such as communication with the nCore API.

The Python module specific to SEE machines is `seeapi.py`. This module is located under Python site packages in `nshield.ipcdaemon.seeapi`. This must be imported as `SEEAPI` to enable critical SEE machine functionality such as communication with the nCore API.

6.3.6. Legacy compatibility

The CodeSafe 5 SDK and nShield 5 HSMs are sufficiently different from previous implementations that legacy applications cannot run with the CodeSafe 5 SDK. For ease of use, the CodeSafe 5 SDK supplies a compatibility layer in the form of headers, files, and libraries to enable legacy applications to be used in nShield 5 HSMs.

Legacy applications require recompilation with new libraries to run on nShield 5 HSMs, see Compatibility layer for legacy SEE machines.

Do not use these compatibility layer libraries, files, and headers to create new SEE machines. They are only supplied to allow legacy applications to be quickly re-compiled and run on nShield 5 HSMs.

6.4. Building new SEE machines with SEElib

An SEE machine is a container image with a complete filesystem which can be loaded onto an CodeSafe 5-enabled HSM as part of a container. The SEElib library enables SEE machines to interface with the nCore API via the IPC daemon.

Source code is compiled using one of the GCC cross-compilers supplied with the CodeSafe SDK. For details of required compiler options, toolchains, makefiles and so on, see the CMake files supplied with the examples, as well as Build and sign example SEE machines on Linux and Build and sign example SEE machines on Windows.

The container image must be signed using the `csadmin` utility tool.

6.4.1. Developer authentication

CodeSafe 5 requires a signed CodeSafe image to run SEE machines on the HSM.

The CodeSafe developer needs to request a developer ID certificate by sending a Certificate Signing Request (CSR) to Entrust support. The tool used to create the CSR is integrated into the HSM software as a subcommand of `csadmin` utility.

For security purposes, a developer keypair must be created and stored within the HSM. In addition, the keypair must be OCS protected to provide authorization control on its use. The developer keypair will be created by `csadmin` if it does not already exist.

After the certificates are received, they are installed on the HSM and are used to sign CodeSafe application images with the `csadmin` tool.

The implementation of this is described in more detail in Sign and deploy CodeSafe 5 SDK apps using csadmin.

6.4.2. Deploying SEE machines

After the code has been compiled, built, and signed, the `csadmin` utility tool is used to deploy the SEE machine. It is used to load the signed CodeSafe application image and then to start the SEE machine. The SEE machine then runs the `entrypoint` including the `main()` function.

For more information on the `csadmin` utility, see Sign and deploy CodeSafe 5 SDK apps using csadmin.

6.4.3. SEE machine initialization requirements

An SEE machine must initialize the `SEElib` before making use of any of the `SEElib` functionality. This is done by calling `SEElib_init()`. It is recommended that this call is made immediately within the `main()` function of an SEE machine.

6.4.4. SEElib Functions

After initialization, SEElib functions can be used to communicate with the nCore API via the IPC daemon. These methods call functions identically to previous CodeSafe versions although the underlying methodology has changed.

6.4.4.1. SEElib_Transact()

To send a command to the nCore API and block waiting for a reply:

```
int SEElib_Transact(struct M_Command *cmd, struct M_Reply *reply)
```

This sends the `cmd` command to the nCore API and waits for the reply to be written to `reply`.

6.4.4.2. SEElib_Submit() / SEElib_Query()

To send a non-blocking command to the nCore API:

```
int SEElib_Submit(M_Command *cmd, M_Reply *reply, PEVENT ev, SEElib_ContextHandle tctx)
```

The `cmd` command is submitted to the nCore API. The transaction listener thread will call EventSet `ev`, if `ev` is non-NULL when the reply returns for this command. The reply is unmarshalled into `reply` and `tctx` is returned to the caller with `SEElib_Query(M_Reply **replyp, SEElib_ContextHandle *tctx_r)`.

Before using the `SEElib_Submit()` method, `SEElib_StartTransactListener()` must have been called to start the transaction listener.

Unlike `SEElib_SubmitCoreJob()`, `SEElib_Submit()` does not block and wait for all other calls to `SEElib_Transact()` to complete.

6.4.4.3. SEElib_SubmitCoreJob / SEElib_GetCoreJobEx()

To submit a job to the nCore API:

```
extern int SEElib_SubmitCoreJob(const unsigned char *data, unsigned int len)
```

To receive a job from the nCore API:

```
extern int SEElib_GetCoreJobEx(unsigned char *buf, M_Word *len_io, unsigned flags)
```

`SEElib_SubmitCoreJob()` is blocking. It waits for the job to be submitted, which includes waiting for existing calls made to `SEElib_Transact()` to be completed. The same is true for `SEElib_GetCoreJobEx()`.

For non-blocking calls, consider using `SEElib_Submit()`.

6.4.4.4. Other SEElib methods

For a comprehensive list of all functionality provided via the SEElib, see: SEE API documentation.

6.4.5. Host/SEE machine communication

The newest CodeSafe 5 implementation simplifies the host/SEE machine connection. Host/SEE machine communication does not need to use SEEJobs or pass through the hardserver and nCore API. Communication between the host-side app and SEE machine is done via TCP/IPv6 networking.

The `ncoreapi` service can only connect to one CodeSafe container at a time.

6.4.5.1. Update Connects running in an IPv4 context

The host side of the CodeSafe 5 examples will only be able to communicate over IPv6. Connects running in an IPv4 context will not be able to run examples without changing how CodeSafe 5 is configured on the Connect. See https://nshielddocs.entrust.com/security-world-docs/hsm-user-guide/hsm-mgmt/codesafe.html for more information.

6.5. Compatibility layer for legacy SEE machines

The CodeSafe 5 SDK provides libraries for developing new SEE machines. It also provides libraries, files, and headers designed for maintaining backwards compatibility with legacy CodeSafe SEE machines.

Do not use the compatibility layer libraries, files, and headers to create new SEE machines. They are only supplied to allow legacy applications to be quickly re-compiled and run on nShield 5 HSMs.

The requirement for a compatibility layer arises from changes made to the overall structure of how CodeSafe 5 SEE machines interact with both the host and with the nCore API:

- Host-SEE machine communication

  In legacy CodeSafe implementations, for older HSMs, communication between a host-side application and an SEE machine would be done via the nCore API using `SEEJobs`. Using the nCore API to relay `SEEJobs` between the host-side and the SEE machine is no longer supported.

Communication via the nCore API has been replaced with direct communication between the host and SEE machine using TCP/UDP socket connections. Optionally, communication can be over an SSH tunnel for security. This allows greater control of the creation, management, and use of connections between the host and SEE machine for developers. It also improves performance as `SEEJobs` no longer have to be sent to the nCore API before being forwarded to the SEE machine.

- SEE machine - nCore API communication

  Communication between the host and SEE machine no longer requires the nCore API as an intermediary. Communication intended to be exclusively between the SEE machine and the nCore API has also changed with the addition of the container IPC daemon. The IPC daemon is provided by Entrust, exists within the container, and maintains connections between the container and the nCore API.

  The IPC daemon forwards commands to the nCore API sent using the `SEElib`. Outside of the addition of the intermediary forwarder, the communication between the SEE machine and the nCore API remains functionally unchanged.

  The `ncoreapi` service can only connect to one CodeSafe container at a time.

The compatibility layer contains two main parts: * `liblegacy_compatibility.a`, the module-side library. * `include-see/legacy-compatibility-host/*`, the host-side compatibility interface.

6.5.1. Module-side compatibility layer

The module-side compatibility layer provides the methods necessary to connect the SEE machine to the host-side application via network connection.

The module-side compatibility layer comprises the `liblegacy_compatibility.a` library. Its install location is:

- **Linux**: `/opt/nfast/c/csd5/lib-ppc64-linux-musl/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\lib-ppc64-linux-musl\`

Legacy SEE machines must be built with `liblegacy_compatibility.a`. When initialized, the module-side compatibility layer opens and maintains a connection between the host-side application and the SEE machine. This allows legacy applications to continue using `SEElib_AwaitJob()` and `SEElib_ReturnJob()` to accept incoming jobs and return them to the host-side application when completed.

6.5.2. Host-side compatibility layer

The host-side compatibility layer provides the methods necessary to connect the host-side application to the SEE machine via network connection.

The host-side compatibility layer comprises the following files:

- `legacy-csee-host-side-compatibility.h` contains all necessary function declarations.
- `legacy-csee-host-side-compatibility.c` contains required host-side function definitions required to connect to and maintain the connection to legacy SEE machines.

Their install location is:

- **Linux**: `/opt/nfast/c/csd5/examples/csee/utils/hostside/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\examples\csee\utils\hostside\`

Legacy host-side applications must be built with `legacy-csee-host-side-compatibility.h` and `legacy-csee-host-side-compatibility.c`. This is done by emulating the connection which was previously created and managed by the hardserver and the nCore API.

`legacy-csee-host-side-compatibility.c` is compiled and added to the `libutil.a` library. Applications should link to it if they need to connect to legacy SEE machines.

6.5.3. Initialize module-side compatibility

Initialize the module-side compatibility layer:

```
extern void SEElib_Legacy_Support_Init(const char* PORT)
```

See Classic SEE (CSEE) examples in Port existing CodeSafe application to CodeSafe 5 for how the module-side legacy support can be initialized to open a socket connection at port `PORT` to communicate between host-side and SEE machines.

6.5.4. Use module-side compatibility

Legacy applications expect incoming messages from the host to be piped from the host to the nCore API via the hardserver. From there, they eventually become accessible within the SEE machine via calls to `SEElib_AwaitJob()` and `SEElib_ReturnJob()`. After the module-side compatibility layer is initialized (see Initialize module-side compatibility), these functions will work exactly as they have in previous CodeSafe applications. No further changes are necessary.

Initializing the compatibility layer functionality via the `SEElib_Legacy_Support_Init()` call allows the compatibility layer to handle incoming and outgoing jobs as would previously have been done by the nCore API. The Classic SEE (CSEE) examples show that the only change made to the SEE machines to allow for backwards compatibility is the initialization of the compatibility layer.

The compatibility layer only supports one client connection at a time while the hardserver can support many.

6.5.5. Initialize host-side application compatibility

Initialize the host-side legacy application to allow connection to the SEE machine, communicating to the host via `PORT`:

```
netsee_initialize_legacy_seejob_support(const char * cseeContainerMachineIPv6, const char *
cseeContainerMachinePort)`
```

Here, `cseeContainerMachinePort` must match the `PORT` initialized by the SEE machine. `cseeContainerMachineIPv6` is the container's IPv6 address. See the execution of CSEE examples in Port existing CodeSafe application to CodeSafe 5 for more information on passing in the IPv6 address of the container.

`netsee_initialize_legacy_seejob_support()` establishes a connection to the SEE machine's container at port `cseeContainerMachinePort`. The compatibility layer maintains this connection and handles the sending of SEEJobs between the host and module SEE machine.

6.5.6. Use host-side application compatibility

The compatibility layer allows host-side application calls to interact with the SEE machine to remain largely unchanged. Some changes to calls are, however, required. These changes, rather than changing how the functions operate, largely serve to remove no longer required elements, such as `NFastApp_Connection`.

- `netsee_transact_legacy_seejob(const M_Command *command, M_Reply *reply, struct NFast_Transaction_Context *tctx)`

  replaces:

  `NFastApp_Transact(NFastApp_Connection conn, struct NFast_Call_Context *cctx, const M_Command *command, M_Reply *reply, struct NFast_Transaction_Context *tctx)`

The NFastApp_Connection and NFast_Call_Context are no longer required and should not be passed in.

- `netsee_simple_transact_legacy_seejob(const M_Command *cmd, M_Reply *reply, int fatal)`

  replaces:

  `simple_transact (NFastApp_Connection nc, M_Command *pcmd, M_Reply *preply, int fatal)`

  

  The NFastApp_Connection is no longer required and should not be passed in.

- `netsee_submit_legacy_seejob(const M_Command *cmd, M_Reply *reply, struct NFast_Transaction_Context *tctx)`

  replaces:

  `NFastApp_Submit(NFastApp_Connection conn, struct NFast_Call_Context *cctx, const M_Command *command, M_Reply *reply, struct NFast_Transaction_Context *tctx)`

  

  The NFastApp_Connection and NFast_Call_Context are no longer required and should not be passed in.

- `netsee_wait_legacy_seejob(M_Reply **replyp, struct NFast_Transaction_Context **tctx)`

  replaces:

  `NFastApp_Wait(NFastApp_Connection conn, struct NFast_Call_Context *cctx, M_Reply **replyp, struct NFast_Transaction_Context **tctx_r)`

  

  The NFastApp_Connection and NFast_Call_Context are no longer required and should not be passed in.

With these changes implemented, legacy host-side applications, when run in conjunction with an SEE machine properly initialized with `liblegacy_compatibility.a`, should function identically to when run in previous implementations of CodeSafe.

This section demonstrated how to use the compatibility layer to quickly bring legacy applications into the new CodeSafe 5 environment. New applications should never be written with the compatibility layer. It is advised that, when possible, a user defined TCP/IPv6 network connection between the host-side application and the SEE machine is

> implemented, rather than using the compatibility layer to transact jobs. However, the compatibility layer does perform this job when no such custom implementation can be made.

7. Sign and deploy CodeSafe 5 SDK apps using csadmin

7.1. Signing CodeSafe images

All CodeSafe images must be signed before they can be loaded on to an HSM. Entrust recommends that you have two signing keys: one that you use to sign CodeSafe images that are still under development, and one that you only use for signing tested CodeSafe images that are ready for deployment. In this guide, the two recommended keys are referred to as the development signing key and the production signing key, however you can name these keys as required by your particular development organisation.

Signed CodeSafe images can be loaded to an HSM if the certificate associated with the signing key is also loaded to that HSM. Therefore you must ensure that the certificates associated with development signing keys are never distributed outside of your development organisation. If you develop CodeSafe images for customers who are not part of your development organisation, you should only send them CodeSafe images that have been signed by, and certificates that are associated with, a production signing key.

You can create as many signing keys as you require. This allows you to use different signing keys to group your CodeSafe images based on whatever criteria you require. For example, you could use different signing keys based on the intended customer or on the functionality of the CodeSafe image.

You must keep track of which key has been used to sign which image and ensure that the end user receives the correct matching certificate and does not receive certificates that they do not require.

The following sections describe the commands used to create the signing keys and certificates followed by a worked example showing the entire process of building, signing, loading, and running a CodeSafe image.

7.2. The csadmin utility tool

The following examples use a Linux machine for the deployment of CodeSafe applications. The same commands can be applied to a Windows machine.

The `csadmin` tool is used to manage CodeSafe images throughout the development and deployment process. It is available as part of the Security World ISO. It must be installed as instructed in Install the CodeSafe 5 SDK on Linux and Install the CodeSafe 5 SDK on Windows.

You must be logged in as an Administrator or a user with local administrator rights to execute `csadmin` commands.

You must have `/opt/nfast/bin` in your `PATH` environment variable to use `csadmin`.

Executing `csadmin` displays the available subcommands:

To view the help text included here while using `csadmin`, run a command or sub-command with the `-h|--help` option.

The `csadmin` tool covers CodeSafe application deployment from both the perspective of a CodeSafe application developer and a CodeSafe application user. The help text displays the complete set of commands available. This document details the commands that are specific to CodeSafe developers. See https://nshielddocs.entrust.com/security-world-docs/utilities/csadmin.html for an overview of the `csadmin` tool and details of the other commands available.

7.2.1. Generate loadable images

CS5 images are generated with `csadmin image generate`. Before generating an image, the CodeSafe 5 SDK must be previously installed. This includes an installation of Python and nfpython suitable to run on the HSM. To display the `generate` operation's usage, execute it with the `--help` option :

```
$ csadmin image generate --help
usage: csadmin image generate [-h] --package-name PACKAGE_NAME --version-str VERSION_STR --entry-point
ENTRY_POINT --network-conf NETWORK_CONF
--packages-conf PACKAGES_CONF --rootdir ROOTDIR [--verbose] CS5FILE

positional arguments:
  CS5FILE               The cs5 file to be handled

optional arguments:
  -h, --help            show this help message and exit
  --package-name PACKAGE_NAME
                        Short name describing the product contents
  --version-str VERSION_STR
                        Version number of this package contents
  --entry-point ENTRY_POINT
                        Full path, within the container, to the entry point application to be executed upon start
  --network-conf NETWORK_CONF
                        Full path, outside the container, to the network config file to be copied into the
container meta data
  --packages-conf PACKAGES_CONF
```

```
                      Full path, outside the container, to the extra packages config file used to copy
additional packages into container rootfs
  --rootdir ROOTDIR     Directory where the contents of the new container are located
  --verbose             Print verbose logs
```

Generating an image requires the name of the CS5 file and the use of the following mandatory command-line arguments:

- `--package-name`
- `--version-str`
- `--entry-point`
- `--network-conf`
- `--packages-conf`
- `--rootdir`

The following items are also required:

- A container directory (not necessarily named "container") that points to what would be the SEE machine's root directory.

  This directory must include any files used by the application, including the entry point program, for example:

  ```
  container/
  ├─── home
  └─── usr
      └─── bin
          └─── entrypoint
  ```

  The container directory can be located anywhere in the host file system. Ensure you pass the full path to the `generate` command via the `--rootdir` argument, as specified in the command usage.

- An entry point program.

  This is the program that runs when the SEE container is started (on launcher start). It must be made executable so it can be launched accordingly. In the previous example, the entry point program is in `container/usr/bin/entrypoint`.

- A network configuration file. (See Example network-conf.json file.)

  The valid range for `container_port` is 1024 - 65535.

- A file with extra packages information. (See Example extra-packages-conf.json file)

7.2.1.1. Example csadmin image generate operation

```
$ csadmin image generate --package-name "MyCodeSafeApp" --entry-point /usr/bin/entrypoint --network-conf network-
conf.json --packages-conf extra-packages-conf.json --version-str 1.0  --rootdir container/ myapp.cs5
INFO: creating content package
INFO: Creating content tar ball
INFO: Creating copy of source file: network-conf.json into dest: cs5_build/meta/network-conf.json
INFO: Creating copy of source file: extra-packages-conf.json into dest: cs5_build/meta/extra-packages-conf.json
INFO: Creating compressed tar ball cs5_build/extra-packages.tar.gz out of cs5_build/extra-packages
INFO: Creating compressed tar ball cs5_build/container.tar.gz out of container/
INFO: Creating uncompressed tar ball content.tar out of cs5_build
INFO: creating cs5 file myapp.cs5
INFO: adding content hash to the package

INFO: File myapp.cs5 was created successfully!
```

`--entry-point` points to the full path of the executable program relative to the container's root.

7.2.1.2. Example extra-packages-conf.json file

```
{
    "packages": [{
            "package": "python",
            "description": "python 3.8 binaries",
            "host_path": "python3/csd5/ppc64/usr/bin",
            "machine_path": "usr/bin",
            "exclude": ""
        },
        {
            "package": "python",
            "description": "python 3.8 libraries",
            "host_path": "python3/csd5/ppc64/usr/lib/python3.8",
            "machine_path": "python3",
            "exclude": ""
        },
        {
            "package": "binaries",
            "description": "binaries for script support 1.0.0",
            "host_path": "c/csd5/rootfs/bin",
            "machine_path": "bin",
            "exclude": ""
        }
    ]
}
```

7.2.1.3. Example network-conf.json file

```
{
    "incoming" : {
        "tcp" : {
            "protos" : [ "ipv6" ], "ports" : [ 8000, 8001, 8888 ]
        }
    },
    "outgoing" : {
        "udp" : {
            "protos" : [ "ipv4" ], "ports" : [ 53 ]
        }
    },
    "ssh_tunnel" : {
             "container_port" : 8000
    }
}
```

7.2.1.4. Example entry point script

```
#!/bin/sh
export PYTHONHOME=/usr/bin
export PYTHONPATH=/usr/lib/python3.8/:/usr/lib/python3.8/lib-dynload:/usr/lib/python3.8/site-packages
python -m http.server --directory / --bind :: 8888
```

7.2.2. Sign images

CodeSafe images are signed with `csadmin image sign`. A signing key must be created

before the CS5 file is signed, because signing must be done using HSM-protected keys.

```
csadmin image sign --help
usage: csadmin image sign [-h] --askeyname ASKEYNAME --devkeyname DEVKEYNAME --devcert DEVCERT [--startdate
STARTDATE] [--expirydate EXPIRYDATE]
                          [--out OUT] [--verbose]
                          CS5FILE

positional arguments:
  CS5FILE               The cs5 file to be signed

options:
  -h, --help            show this help message and exit
  --askeyname ASKEYNAME
                        Name (ident) of the application signing key
  --devkeyname DEVKEYNAME
                        Name (ident) of the developer signing key
  --devcert DEVCERT     The signed developer certificate PEM file
  --startdate STARTDATE
                        Start of validity period for the signed ASK cert in Unix time (default: no start date)
  --expirydate EXPIRYDATE
                        End of validity period for the signed ASK cert in Unix time (default: no expiration date)
  --out OUT             Name of the output file. If not specified, the cs5 file is overwritten.
  --verbose             Print verbose logs
```

For more information, see Signing CodeSafe images.

7.2.3. Create a developer ID certificate

Developer ID certificates are created with `csadmin ids create`. This command generates a developer ID key with the given name (if it doesn't exist already) and a certificate signing request so a certificate can be generated (see Signing CodeSafe images):

```
$ csadmin ids create --help
usage: csadmin ids create [-h] --keyname KEYNAME [-m MODULE] --x509cname COMMON_NAME [--x509country COUNTRY]
                          [--x509province STATE_OR_PROVINCE] [--x509locality LOCALITY] --x509org ORGANIZATION [--
x509orgunit ORGANIZATIONAL_UNIT] [--verbose]

options:
  -h, --help            show this help message and exit
  --keyname KEYNAME     Name for the certificate's key.
  -m MODULE, --module MODULE
                        Module to generate the key with.
  --x509cname COMMON_NAME
                        The CN part of the key's DN.
  --x509country COUNTRY
                        The C part of the key's DN.
  --x509province STATE_OR_PROVINCE
                        The ST part of the key's DN.
  --x509locality LOCALITY
                        The L part of the key's DN.
  --x509org ORGANIZATION
                        The O part of the key's DN.
  --x509orgunit ORGANIZATIONAL_UNIT
                        The OU part of the key's DN.
  --verbose             Print verbose logs
```

7.3. Example CodeSafe developer process

The examples in this chapter show how various `csadmin` commands can be used to create a signed CodeSafe image for deployment. For details of the `csadmin` tool (See The csadmin utility tool)

7.3.1. Create developer ID keys

To sign CodeSafe images, you must create a developer ID for your development organisation and obtain a matching certificate from Entrust. You can obtain a certificate by creating a Certificate Signing Request (CSR) file and sending it to Entrust Support who will process the CSR and return a signed certificate to you.

Entrust strongly recommend that you create at least two developer IDs: a 'development' ID for signing CodeSafe images that are still in development, and a 'production' ID for signing images that are ready to be deployed.

The `csadmin ids create` command provides the functionality to generate a developer ID key if it does not already exist, as well as the CSR file in a single step.

Keep track of which certificate matches each developer ID key. When you send a signed CodeSafe image to a customer you will need to also send them the matching certificate for them to be able to load the image on their HSM.

The developer ID keys only need to be created once. The certificates matching them have a limited validity period and will need to be refreshed before they expire.

When you refresh a certificate you must send it to anyone who received a copy of a SEE machine that is signed by the key matching that certificate. Users of SEE machines require a valid certificate every time they start the SEE machine.

To refresh a certificate, use the `csadmin ids create` command with an existing key. This creates a CSR file for the existing key, which should be sent to Entrust Support who will process the CSR and return a new signed certificate.

The integrity of the signing process relies on the procedural steps being followed to secure a CodeSafe application image.

For this reason, developer ID keys are OCS protected and therefore to sign a CodeSafe

application a quorum of OCS cards and associated passphrases must be available for the signing.

Only use your 'production' developer ID key to sign fully tested CodeSafe images that you know to be ready for deployment.

7.3.1.1. Generate an HSM-protected developer ID key and CSR

```
csadmin ids create --keyname developerid --x509cname developer.entrust.com --x509country US --x509province
Minnesota --x509locality Shakopee --x509org "CodeSafe App Development" --x509orgunit "Entrust CodeSafe"

Generate key 'testdeveloperkey' ...

Loading `TestOCS':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: empty
Card reading complete.

OK
Generate a CSR in 'testdeveloperkey.csr' ...
OK
Created CSR file 'testdeveloperkey.csr'. Please send it to Entrust Support
```

This creates the CSR file in the location where the command was run.

`keyname` must conform with character set restrictions. For more information, see `ident` in the Key properties table.

This developer ID creation was done with `TestOCS`, quorum of 1/1. Exact output might vary slightly with different OCS quorums.

Send the resulting CSR to customer support to be signed by Entrust.

7.3.2. Load your certificate

When you receive your signed certificate chain back from Entrust Support, load the developer ID certificate chain in the HSM using `csadmin ids add`.

You can use `csadmin ids list` to view the loaded certificate.

```
$ csadmin ids add entrust_developerid_cert_chain.pem
FEDC-BA09-8765        SUCCESS
$ csadmin ids list
FEDC-BA09-8765        SUCCESS
Certificates:
{'serialNumber': '1', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust CodeSafe,
Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'abcdef12345678900987654321fedcbaabcdef12', 'authKeyid': '0987654321fedcbaabcdef123456789009876543', 'notBefore':
'2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
{'serialNumber': '2', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust CodeSafe,
Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'1234567890abcdeffedcba098765432112345678', 'authKeyid': 'fedcba09876543211234567890abcdeffedca098', 'notBefore':
'2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
```

7.3.2.1. Generate an Application Signing Key (ASK) with generatekey

This generates a simple ECDSA NIST521P key.

The following example specifies the key to be protected with an OCS.

```
/opt/nfast/bin/generatekey --batch --module=1 simple type=ECDSA curve=NISTP521 ident=ask plainname=ask
protect=token
```

7.3.2.2. Sign the CodeSafe image

This example signs a CodeSafe application called `hello.cs5`:

```
csadmin image sign --askeyname ask --devkeyname developerid --devcert ~/ca/developerid_cert.pem --out ~/hello-
signed.cs5 ~/hello.cs5
```

8. Build and sign example SEE machines on Linux

8.1. Build module-side C examples

1. Create an empty directory to build the module side examples into, for example:

```
mkdir ~/buildmodule/
```

2. Navigate to the empty directory:

```
cd ~/buildmodule/
```

3. Build the module side examples with `cmake` using the following commands:

```
cmake -DCMAKE_TOOLCHAIN_FILE=/opt/nfast/c/csd5/cmake/codesafe-toolchain-nshield5-csee.cmake
/opt/nfast/c/csd5/examples/

cmake --build .
```

Successful builds create `.cs5` images for each example. For example, the classic SEE `Hello` example has a `.cs5` image at `~/buildmodule/n5/csee/hello/module/hello.cs5`.

8.2. Building Host Side C Examples

1. Create an empty directory to build the host-side clients for the SEE machines, for example:

```
mkdir ~/buildhost/
```

2. Navigate to the directory where the host-side examples will be built:

```
cd ~/buildhost/
```

3. Build the host-side examples with `cmake` using the following commands:

```
cmake /opt/nfast/c/csd5/examples/

cmake --build .
```

Successful builds create executable host-side clients for each example. For example, the

classic SEE `Hello` example has an executable program at `~/buildhost/n5/csee/hello/host/hello`.

8.3. Build CS5 Images for Python Examples

1. Create an empty directory to build the Python examples into, for example:

```
mkdir ~/build_python
```

2. Navigate to the empty directory:

```
cd ~/build_python/
```

3. Build the examples with `cmake` using the following commands:

```
cmake /opt/nfast/python3/csd5/examples

cmake --build .
```

Successful builds create `.cs5` images and executable host-side clients for each example. For example, the `hello_tcp` example has a `.cs5` image at `~/build_python/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp.cs5` and the executable program is located at `~/build_python/n5/netsee/helloworld_tcp/hostside/helloworld_host_tcp.py`.

8.4. Sign CodeSafe Images

1. Use `csadmin ids create` to generate the developer ID key, if it does not already exist, as well as the CSR file in a single step. If the key already exists, it only generates the CSR.

```
csadmin ids create --keyname developerid --x509cname developer.entrust.com --x509country US --x509province
Minnesota --x509locality Shakopee --x509org "Entrust CodeSafe" --x509orgunit "Entrust CodeSafe"

Generate key 'testdeveloperkey' ...

Loading `TestOCS':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: empty
Card reading complete.

OK
Generate a CSR in 'testdeveloperkey.csr' ...
OK
Created CSR file 'testdeveloperkey.csr'. Please send it to Entrust Support
```

This creates the CSR file in the location where the command was run. This developer ID creation was done with `TestOCS`, quorum of 1/1. Exact output might vary slightly with different OCS quorums.

2. Send the CSR to customer support to be signed by Entrust. You must obtain the signed developer ID certificate in order to sign and load an application.

For more detailed information on Developer IDs and CSRs, see Sign and deploy CodeSafe 5 SDK apps using csadmin.

3. Use `nfast generatekey` to generate a simple ECDSA NIST521P application signing key (ASK). The following example specifies the key to be protected by the module. However, end users are encouraged to protect the key with an OCS.

```
/opt/nfast/bin/generatekey --batch --module=1 simple type=ECDSA curve=NISTP521 ident=ask plainname=ask
protect=module
```

4. Sign the CodeSafe image, for example:

```
csadmin image sign --askeyname ask --devkeyname developerid --devcert ~/ca/developerid_cert.pem --out
/tmp/hello-signed.cs5 ~/ca/hello.cs5
```

Additional examples are provided later in this chapter.

5. Use `csadmin ids add` to install the developer ID certificate chain from Entrust.

You can use `csadmin ids list` to view the loaded certificate.

```
$ csadmin ids add entrust_developerid_cert_chain.pem
FEDC-BA09-8765        SUCCESS
$ csadmin ids list
FEDC-BA09-8765        SUCCESS
Certificates:
{'serialNumber': '1', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust
CodeSafe, Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'abcdef12345678900987654321fedcbaabcdef12', 'authKeyid': '0987654321fedcbaabcdef123456789009876543',
'notBefore': '2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
{'serialNumber': '2', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust
CodeSafe, Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'1234567890abcdeffedcba098765432112345678', 'authKeyid': 'fedcba0987654321123456789abcdeffedca098',
'notBefore': '2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
```

8.5. Run NetSEE examples

`NetSEE` examples communicate between the client and SEE machine directly through a TCP/IPv6 network connection to the container, unlike legacy applications, such as for Solo XC or Solo+, which communicate through the hardserver to the nCore API.

8.5.1. helloworld_tcp

To execute the helloworld TCP example that opens a socket within the container and uses the connection to transact a "helloworld" message:

1. Sign the `.cs5` image using `devcert` and `askeys`:

```
csadmin image sign --askeyname ask --devkeyname developerid --devcert ~/ca/developerid_cert.pem --out
~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp-signed.cs5
~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp.cs5
```

2. Load the signed `.cs5` image using `csadmin load`:

```
sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp-
signed.cs5
```

The output of `csadmin load` contains the UUID of the loaded container. This UUID will be required for starting the container. The UUID can always be retrieved from the output of `csadmin list`.

3. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

`csadmin list` lists the UUIDs of all containers. The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

4. Run the host-side application.

   The host-side application takes three positional arguments, the IPv6 address of the container, the port number, and the message to send to the container. The port number used by this example is 8888 by default. The message can be any string of valid characters.

```
~/buildhost/n5/netsee/helloworld_tcp/hostside/helloworld_host_tcp ffff::fff:ffff:ffff:ffff%nshield0 8888
hello_module
```

   Expected output:

```
nseeContainerMachineIP=ffff::fff:ffff:ffff:ffff%nshield0
nseeContainerMachinePort=8888
mesg=hello_module
Successful Connection to Socket...
```

```
Host>Sending TCP Message-->hello_module
Host>Hello World From HSM!
```

The IPv6 address is link-local and requires the zone index to be appended (typically `%nshield0`).

8.5.1.1. helloworld_tcp for nShield 5c

The process is the same as the 5s example, but the host-side application command will differ. Instead of IPv6, you can use the Connect's IPv4 address:

The examples for the nShield 5c work similarly to the 5s module, but the IP addresses and ports refer to the 5c Connect network. Similarly, for the TCP example, you can use the Connect's IPv4 address:

```
~/buildhost/n5/netsee/helloworld_tcp/hostside/helloworld_host_tcp 192.168.1.100 8888 hello_module
```

Example output:

```
nseeContainerMachineIP=192.168.1.100
nseeContainerMachinePort=8888
mesg=hello_module
Successful Connection to Socket...
Host>Sending TCP Message-->hello_module
Host>Hello World From HSM!
```

8.5.2. helloworld_udp

To execute the helloworld UDP example that opens a socket within the container and uses the connection to transact a "helloworld" message:

1. Sign the `.cs5` image using `devcert` and `askeys`:

   ```
   csadmin image sign --askeyname ask --devkeyname developerid --devcert ~/ca/developerid_cert.pem --out
   ~/buildmodule/n5/netsee/helloworld_udp/module/helloworld_mod_udp-signed.cs5
   ~/buildmodule/n5/netsee/helloworld_udp/module/helloworld_mod_udp.cs5
   ```

2. Load the signed container using `csadmin load`:

   ```
   sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/netsee/helloworld_udp/module/helloworld_mod_udp-
   signed.cs5
   ```

   Example output:

   ```
   FEDC-BA09-8765: Uploading ~/buildmodule/n5/netsee/helloworld_udp/module/helloworld_mod_udp-signed.cs5
   ```

```
FEDC-BA09-8765: creating machine
FEDC-BA09-8765        SUCCESS
UUID: fedcba09-8765-4321-1234-567890abcdef
```

The output of `csadmin load` contains the UUID of the loaded container. This UUID will be required for starting the container. The UUID can always be retrieved from the output of `csadmin list`.

3. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
IP ADDRESS: ffff::fff:ffff:ffff:ffff
```

`csadmin list` will list the UUIDs of all containers. The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

4. Run the host-side application.

The host-side application takes three positional arguments, the IPv6 address of the container, the port number, and the message to send to the container. The port number used by this example is 8888 by default. The message can be any string of valid characters.

```
~/buildhost/n5/netsee/helloworld_udp/hostside/helloworld_host_udp ffff::fff:ffff:ffff:ffff%nshield0 8888
hello_module
```

Example output:

```
nseeContainerMachineIP=ffff::fff:ffff:ffff:ffff%nshield0
nseeContainerMachinePort=8888
mesg=hello_module
Successful Connection to Socket...
Host>Sending UDP Message-->hello_module
Host>Hello World From HSM!
```

The IPv6 address is link-local and requires the zone index to be appended (typically `%nshield0`).

8.5.2.1. helloworld_udp for 5c

The process is the same as the 5s example, but the host-side application command will differ. Instead of IPv6, you can use the Connect's IPv4 address:

The examples for the nShield 5c work similarly to the 5s module, but the IP addresses and ports refer to the 5c Connect network.

```
~/buildhost/n5/netsee/helloworld_udp/hostside/helloworld_host_udp 192.168.1.100 8888 hello_module
```

Example output:

```
nseeContainerMachineIP=192.168.1.100
nseeContainerMachinePort=8888
mesg=hello_module
Successful Connection to Socket...
Host>Sending UDP Message-->hello_module
Host>Hello World From HSM!
```

8.6. Run NetSEE examples via SSH tunnel

`NetSEE` examples communicate between the client and SEE machine directly through a TCP/IPv6 network connection to the container, unlike legacy applications, such as for Solo XC or Solo+, which communicate through the hardserver to the nCore API.

On the nShield 5c network, the `SSHD listening address` may be an `IPv4` address instead of `IPv6`. Adjustments to the steps below may be needed to accommodate this.

8.6.1. helloworld_tcp via SSH Tunnel

To execute the helloworld TCP example via an SSH Tunnel that opens a socket within the container and uses the connection to transact a "helloworld" message:

1. Create an SSHD key for the hello example:

   ```
   mkdir ~/examplekeys/
   ssh-keygen -t ecdsa -f ~/examplekeys/helloworld_tcp_ecdsa_key
   ```

2. Modify the `network-conf.json` of the `helloworld_tcp` example to support SSH tunneling, for example:

   ```
   cat ~/buildmodule/n5/netsee/helloworld_tcp/module/network-conf.json
   {
           "incoming": {
   ```

```
                "tcp":
                {
                        "protos": ["ipv6"],
                        "ports": [8888]
                }
        },
        "outgoing" : {
                "tcp" :
                {
                       "protos": ["ipv6"],
                       "ports": []
                }
        },
        "ssh_tunnel" : {
                 "container_port" : 8888
        }
}
```

When the container server app accepts a client connection on the specified incoming port (for example `8888`), it designates and responds to the client on an ephemeral port in the range `[32768-60999]` as the outgoing port. This port does not have to be defined in the `network-conf.json`.

3. Rebuild the `.cs5` image with the updated `network-conf.json` so the loaded container will allow SSH tunneling:

```
sudo /opt/nfast/bin/csadmin image generate --package-name "helloworld_tcp" --entry-point
/usr/bin/entrypoint --network-conf ~/buildmodule/n5/netsee/helloworld_tcp/module/network-conf.json
--packages-conf ~/buildmodule/n5/netsee/helloworld_tcp/module/extra-packages-conf.json --version-str 1.0
--rootdir ~/buildmodule/n5/netsee/helloworld_tcp/module/container/
~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp.cs5
```

Most paths used in generating the new image are paths to the file locations on the host that is building the image However, the `--entry-point` path is the absolute path to the `entrypoint` file within the container and should be `/usr/bin/entrypoint`, not `~/buildmodule/n5/netsee/helloworld_tcp/module/container/usr/bin/entrypoint`.

4. Sign the new `.cs5` image using `devcert` and `askeys`:

```
sudo /opt/nfast/bin/csadmin image sign --askeyname ask --devkeyname developerid --devcert
~/ca/developerid_cert.pem --out ~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp-signed.cs5
~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp.cs5
```

5. Load the signed container using `csadmin load`:

```
sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/netsee/helloworld_tcp/module/helloworld_mod_tcp-
signed.cs5
```

The output of `csadmin load` contains the UUID of the loaded container. This UUID will be required for starting the container and managing the SSHD keys of the container.

The UUID can always be retrieved from the output of `csadmin list`.

6. Load the public key created earlier (`helloworld_tcp_ecdsa_key`) to the container using `csadmin sshd setclient`:

```
sudo /opt/nfast/bin/csadmin sshd keys setclient --uuid fedcba09-8765-4321-1234-567890abcdef --keyfile
~/examplekeys/helloworld_tcp_ecdsa_key.pub
```

7. Enable SSH tunneling on the container:

```
sudo /opt/nfast/bin/csadmin sshd state enable --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
SSHD PORT: 6789
LISTENING ADDRESS: aaaa::aa:aaaa:aaaa:aaaa
```

The output of `sshd state enable` contains the SSHD Port number and the listening address of the container SSHD.

8. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

`csadmin list` lists the UUIDs of all containers. The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

9. Setup the SSH tunnel on the host:

Run `csadmin sshd state get` and collect the following information:

- Container tunnel address (`ffff::fff:ffff:ffff:ffff`)
- Container port (`8888`)
- SSHD port (`6789`)
- SSHD listening address (`aaaa::aa:aaaa:aaaa:aaaa`)

On nShield Connect the `SSHD listening address` may be an `IPv4` or `IPv6` address

Next, choose a local IP address and port number through which to access the tunnel. Typically localhost is chosen as the local IP address (`127.0.0.1` or `[::1]`)

The SSH tunnel command is formatted as follows:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L LOCAL_IP:LOCAL_PORT:[TUNNEL_ADDRESS%lxcbr0]:CONTAINER_PORT
-f -N -p SSHD_PORT launcher@LISTENING_ADDRESS
```

Using the example data:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L [::1]:8888:[ffff::fff:ffff:ffff:ffff%lxcbr0]:8888 -f -N -p
6789 launcher@aaaa::aa:aaaa:aaaa:aaaa%nshield0
```

When using nShield 5s the IPv6 address is link-local and requires the zone index to be appended (typically `%nshield0`). If you are working with a 5c network, replace the IPv6 address with the appropriate nShield5c network address (IPv4 or IPv6) for your configuration.

10. Run the host-side application.

    The host-side application takes three positional arguments, the IPv6 address set up in the forwarding step `[::1]`, the port number, and the message to send to the container. The port number used by this example is 8888 by default. The message can be any string of valid characters.

    ```
    ~/buildhost/n5/netsee/helloworld_tcp/hostside/helloworld_host_tcp ::1 8888 hello_module
    ```

    Expected Output:

    ```
    nseeContainerMachineIP=::1
    nseeContainerMachinePort=8888
    mesg=hello_module
    Successful Connection to Socket...
    Host>Sending TCP Message-->hello_module
    Host>Hello World From HSM!
    ```

8.7. Run CSEE examples via SSH tunnel

The Classic SEE (CSEE) examples are legacy examples modified to run with CodeSafe 5 to demonstrate use of the compatibility layer. These examples are identical to examples provided with previous iterations of nShield HSMs and CodeSafe. This section describes running the CSEE examples using an SSH Tunnel

8.7.1. hello via SSH Tunnel

This section describes executing the legacy `hello` example using the compatibility layer via an SSH Tunnel. The CSEE hello example operates functionally identically to previous hello examples for Solo XC and Solo+.

The hello example sends a string from the host to the module. The module converts the string to uppercase and returns the string to the host.

1. Generate an input file containing a character string to be sent to the module.

   ```
   echo UPPERCASElowercase > ~/inputfile
   ```

   This input file has both uppercase and lowercase characters.

2. Generate an SSHD key for the hello example:

   ```
   mkdir ~/examplekeys/
   ssh-keygen -t ecdsa -f ~/examplekeys/hello_ecdsa_key
   ```

3. Modify the `network-conf.json` of the hello example to configure SSH tunneling, for example:

   ```
   cat ~/buildmodule/n5/csee/hello/module/network-conf.json
   {
           "incoming": {
                   "tcp":
                   {
                           "protos": ["ipv6"],
                           "ports": [8888]
                   }
           },
           "outgoing" : {
                   "tcp" :
                   {
                          "protos": ["ipv6"],
                          "ports": []
                   }
           },
           "ssh_tunnel" : {
                    "container_port" : 8888
           }
   }
   ```

   

   When the container server app accepts a client connection on the specified incoming port (for example `8888`), it designates and responds to the client on an ephemeral port in the range `[32768-60999]` as the outgoing port. This port does not have to be defined in the `network-conf.json`.

4. Rebuild the `.cs5` image with the updated `network-conf.json` so the loaded container will allow SSH tunneling:

```
sudo /opt/nfast/bin/csadmin image generate --package-name "hello" --entry-point /usr/bin/entrypoint
--network-conf ~/buildmodule/n5/csee/hello/module/network-conf.json --packages-conf
~/buildmodule/n5/csee/hello/module/extra-packages-conf.json --version-str 1.0  --rootdir
~/buildmodule/n5/csee/hello/module/container/ ~/buildmodule/n5/csee/hello/module/hello.cs5
```

Most paths used in generating the new image are paths to the file locations on the host that is building the image. However, the `--entry-point` path is the absolute path to the `entrypoint` file within the container and should be `/usr/bin/entrypoint`, not `~/buildmodule/n5/csee/hello/module/container/usr/bin/entrypoint`.

5. Sign the `.cs5` image using `devcert` and `askeys`:

```
sudo /opt/nfast/bin/csadmin image sign --askeyname ask --devkeyname developerid --devcert
~/ca/developerid_cert.pem --out ~/buildmodule/n5/csee/hello/module/hello-signed.cs5
~/buildmodule/n5/csee/hello/module/hello.cs5
```

6. Load the signed container using `csadmin load`:

```
sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/csee/hello/module/hello-signed.cs5
```

Example output:

```
FEDC-BA09-8765: Uploading ~/buildmodule/n5/csee/hello/module/hello-signed.cs5
FEDC-BA09-8765: creating machine
FEDC-BA09-8765        SUCCESS
UUID: fedcba09-8765-4321-1234-567890abcdef
```

The output of `csadmin load` contains the UUID of the loaded container. This UUID will be required for starting the container and managing the SSHD keys of the container. The UUID can always be retrieved from the output of `csadmin list`.

7. Load the public key created earlier (`hello_ecdsa_key`) to the container using `csadmin sshd setclient`:

```
sudo /opt/nfast/bin/csadmin sshd keys setclient --uuid fedcba09-8765-4321-1234-567890abcdef --keyfile
~/examplekeys/hello_ecdsa_key.pub
```

8. Enable SSH tunneling on the container:

```
sudo /opt/nfast/bin/csadmin sshd state enable --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
SSHD PORT: 6789
```

```
LISTENING ADDRESS: aaaa::aa:aaaa:aaaa:aaaa
```

The output of `sshd state enable` contains the SSHD port number and the listening address of the container SSHD.

9. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
IP ADDRESS: ffff::fff:ffff:ffff:ffff
```

`csadmin list` lists the UUIDs of all containers. The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

10. Setup the SSH tunnel on the host:

Run `csadmin sshd state get` and collect the following information:

- Container tunnel address (`ffff::fff:ffff:ffff:ffff`)
- Container port (`8888`)
- SSHD port (`6789`)
- SSHD listening address (`aaaa::aa:aaaa:aaaa:aaaa`)

On nShield Connect the `SSHD listening address` may be an `IPv4` or `IPv6` address

Next, choose a local IP address and port number through which to access the tunnel. Typically localhost is chosen as the local IP address (`127.0.0.1` or `[::1]`)

The SSH tunnel command is formatted as follows:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L LOCAL_IP:LOCAL_PORT:[TUNNEL_ADDRESS%lxcbr0]:CONTAINER_PORT
-f -N -p SSHD_PORT launcher@LISTENING_ADDRESS
```

Using the example data:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L [::1]:8888:[ffff::fff:ffff:ffff:ffff%lxcbr0]:8888 -f -N -p
6789 launcher@aaaa::aa:aaaa:aaaa:aaaa%nshield0
```

When using nShield 5s the IPv6 address is link-local and requires

> the zone index to be appended (typically `%nshield0`).

11. Run the host-side application.

    The host-side application takes one required positional argument, and three required optional arguments. The required optional arguments are the IPv6 address set up in the forwarding step `[::1]` (`--ipv6`), the UUID of the container (`--uuid`), and the file path to the signed container image (`--cs5`). The required positional argument is the input file containing a string to convert to uppercase on the module.

    ```
    ~/buildhost/n5/csee/hello/hostside/hello --uuid fedcba09-8765-4321-1234-567890abcdef --ipv6 ::1 --cs5
    ~/buildmodule/n5/csee/hello/module/hello-signed.cs5 ~/inputfile
    ```

    Example output:

    ```
    Worldid: 0x1234abcd
    UPPERCASELOWERCASE
    ```

    The module has received the input string `UPPERCASElowercase` and has converted and returned it as a fully uppercase string `UPPERCASELOWERCASE`.

8.7.2. tickets via SSH tunnel

This section describes executing the legacy `tickets` example using the compatibility layer via an SSH Tunnel. The CSEE tickets example operates functionally identically to previous tickets examples for Solo XC, Solo+. The tickets example serves to demonstrate cryptographic functionality by encrypting and having the module decrypt a user-provided string.

1. Generate a simple RSA key to encrypt with:

    ```
    sudo /opt/nfast/bin/generatekey --module=1 simple type=RSA pubexp=3 ident=encryptionkeytickets
    plainname=encryptionkeytickets protect=module nvram=no size=2048
    ```

2. Generate an SSHD key for the tickets example:

    ```
    mkdir ~/examplekeys/
    ssh-keygen -t ecdsa -f ~/examplekeys/tickets_ecdsa_key
    ```

3. Modify the `network-conf.json` of the tickets example to configure SSH tunneling, for example:

    ```
    cat ~/buildmodule/n5/csee/tickets/module/network-conf.json
    ```

```
{
        "incoming": {
                "tcp":
                {
                        "protos": ["ipv6"],
                        "ports": [8888]
                }
        },
        "outgoing" : {
                "tcp" :
                {
                       "protos": ["ipv6"],
                       "ports": []
                }
        },
        "ssh_tunnel" : {
                 "container_port" : 8888
        }
}
```

When the container server app accepts a client connection on the specified incoming port (for example `8888`), it designates and responds to the client on an ephemeral port in the range `[32768-60999]` as the outgoing port. This port does not have to be defined in the `network-conf.json`.

4. Rebuild the `.cs5` image with the updated `network-conf.json`:

```
sudo /opt/nfast/bin/csadmin image generate --package-name "tickets" --entry-point /usr/bin/entrypoint
--network-conf ~/buildmodule/n5/csee/tickets/module/network-conf.json --packages-conf
~/buildmodule/n5/csee/tickets/module/extra-packages-conf.json --version-str 1.0  --rootdir
~/buildmodule/n5/csee/tickets/module/container/ ~/buildmodule/n5/csee/tickets/module/seetickets.cs5
```

   Most paths used in generating the new image are paths to the file locations on the host that is building the image. However, the `--entry-point` path is the absolute path to the `entrypoint` file within the container and should be `/usr/bin/entrypoint`, not `~/buildmodule/n5/csee/tickets/module/container/usr/bin/entrypoint`.

5. Sign the `.cs5` image using `devcert` and `askeys`:

```
sudo /opt/nfast/bin/csadmin image sign --askeyname ask --devkeyname developerid --devcert
~/ca/developerid_cert.pem --out ~/buildmodule/n5/csee/tickets/module/seetickets-signed.cs5
~/buildmodule/n5/csee/tickets/module/seetickets.cs5
```

6. Load the signed container using `csadmin load`:

```
sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/csee/tickets/module/seetickets-signed.cs5
```

   Example output:

```
FEDC-BA09-8765: Uploading ~/buildmodule/n5/csee/tickets/module/seetickets-signed.cs5
FEDC-BA09-8765: creating machine
FEDC-BA09-8765        SUCCESS
UUID: fedcba09-8765-4321-1234-567890abcdef
```

The output of `csadmin load` contains the UUID of the loaded container. This UUID will be required for starting the container and managing the SSHD keys of the container. The UUID can also be retrieved from the output of `csadmin list`.

7. Load the public key created earlier (`tickets_ecdsa_key`) to the container using `csadmin sshd setclient`:

```
sudo /opt/nfast/bin/csadmin sshd keys setclient --uuid fedcba09-8765-4321-1234-567890abcdef --keyfile
~/examplekeys/tickets_ecdsa_key.pub
```

8. Enable SSH tunneling on the container:

```
sudo /opt/nfast/bin/csadmin sshd state enable --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
SSHD PORT: 6789
LISTENING ADDRESS: aaaa::aa:aaaa:aaaa:aaaa
```

The output of `sshd state enable` contains the SSHD Port number and the listening address of the container sshd.

9. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
IP ADDRESS: ffff::fff:ffff:ffff:ffff
```

The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

10. Setup the SSH tunnel on the host:

Run `csadmin sshd state get` and collect the following information:

- Container tunnel address (`ffff::fff:ffff:ffff:ffff`)
- Container port (`8888`)
- SSHD port (`6789`)
- SSHD listening address (`aaaa::aa:aaaa:aaaa:aaaa`)

On nShield Connect the `SSHD listening address` may be an `IPv4` or `IPv6` address

Next, choose a local IP address and port number through which to access the tunnel. Typically localhost is chosen as the local IP address (`127.0.0.1` or `[::1]`)

The SSH tunnel command is formatted as follows:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L LOCAL_IP:LOCAL_PORT:[TUNNEL_ADDRESS%lxcbr0]:CONTAINER_PORT
-f -N -p SSHD_PORT launcher@LISTENING_ADDRESS
```

Using the example data:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L [::1]:8888:[ffff::fff:ffff:ffff:ffff%lxcbr0]:8888 -f -N -p
6789 launcher@aaaa::aa:aaaa:aaaa:aaaa%nshield0
```

When using nShield 5s the IPv6 address is link-local and requires the zone index to be appended (typically `%nshield0`).

11. Run the host-side application.

    The host-side application takes three required optional arguments. The required optional arguments are the IPv6 address set up in the forwarding above `[::1]` (`--ipv6`), the UUID of the container (`--uuid`), and the file path of the signed `.cs5` image (`--cs5`). The host-side also accepts the encryption key created earlier as an optional argument (`--key`).

    ```
    ~/buildhost/n5/csee/tickets/hostside/hosttickets --uuid fedcba09-8765-4321-1234-567890abcdef --ipv6 ::1
    --cs5 ~/buildmodule/n5/csee/tickets/module/seetickets-signed.cs5 --key simple,encryptionkeytickets
    ```

12. When prompted, enter a string to encrypt (for example, `testencryption`) and press **Return**:

    ```
    Enter string to be encrypted (256 characters maximum): testencryption
    ```

    The host encrypts the message then the module decrypts it and returns it in plain text format.

Example output:

```
HostSide> Loading security world key (simple,encryptionkeytickets)
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:
"testencryption"
HostSide> Thank you for watching. The end.
```

8.7.3. benchmark via SSH tunnel

This section describes executing the legacy `benchmark` example using the compatibility layer via an SSH tunnel. The CSEE benchmark example operates functionally identically to previous benchmark examples for Solo XC and Solo+. The benchmark example will transact asynchronously with the module running multiple threads processing transactions. The benchmark example will output transactions/second data every second.

1. Generate a simple key for signing a ticket in the bm-machine on the module:

```
sudo /opt/nfast/bin/generatekey --module=1 simple type=RSA pubexp=3 ident=signingkeybenchmark
plainname=signingkeybenchmark protect=module nvram=no size=2048
```

2. Generate an SSHD key for the benchmark example:

```
mkdir ~/examplekeys/
ssh-keygen -t ecdsa -f ~/examplekeys/benchmark_ecdsa_key
```

3. Modify the `network-conf.json` of the benchmark example to configure SSH tunneling, for example:

```
cat ~/buildmodule/n5/csee/benchmark/module/network-conf.json
{
        "incoming": {
                "tcp":
                {
                        "protos": ["ipv6"],
                        "ports": [8888]
                }
        },
        "outgoing" : {
                "tcp" :
                {
                       "protos": ["ipv6"],
                       "ports": []
                }
        },
        "ssh_tunnel" : {
                 "container_port" : 8888
        }
```

```
}
```

When the container server app accepts a client connection on the specified incoming port (8888), it designates and responds to the client on an ephemeral port in the range [32768-60999] as the outgoing port. This port does not have to be defined in the network-conf.json.

4. Rebuild the .cs5 image with the updated network-conf.json:

```
sudo /opt/nfast/bin/csadmin image generate --package-name "bm-machine" --entry-point /usr/bin/entrypoint
--network-conf ~/buildmodule/n5/csee/benchmark/module/network-conf.json --packages-conf
~/buildmodule/n5/csee/benchmark/module/extra-packages-conf.json --version-str 1.0  --rootdir
~/buildmodule/n5/csee/benchmark/module/container/ ~/buildmodule/n5/csee/benchmark/module/bm-machine.cs5
```

   Most paths used in generating the new image are paths to the file locations on the host that is building the image. However, the --entry-point path is the absolute path to the entrypoint file within the container and should be /usr/bin/entrypoint, not ~/buildmodule/n5/csee/benchmark/module/container/usr/bin/entrypoint.

5. Sign the .cs5 image using devcert and askeys:

```
sudo /opt/nfast/bin/csadmin image sign --askeyname ask --devkeyname developerid --devcert
~/ca/developerid_cert.pem --out ~/buildmodule/n5/csee/benchmark/module/bm-machine-signed.cs5
~/buildmodule/n5/csee/benchmark/module/bm-machine.cs5
```

6. Load the signed container using csadmin load:

```
sudo /opt/nfast/bin/csadmin load ~/buildmodule/n5/csee/benchmark/module/bm-machine-signed.cs5
```

   Example output:

```
FEDC-BA09-8765: Uploading ~/buildmodule/n5/csee/benchmark/module/bm-machine-signed.cs5
FEDC-BA09-8765: creating machine
FEDC-BA09-8765        SUCCESS
UUID: fedcba09-8765-4321-1234-567890abcdef
```

   The output of csadmin load contains the UUID of the loaded container. This UUID will be required for starting the container and managing the SSHD keys of the container. The UUID can always be retrieved from the output of csadmin list.

7. Load the public key created earlier (benchmark_ecdsa_key) to the container using csadmin sshd setclient:

```
sudo /opt/nfast/bin/csadmin sshd keys setclient --uuid fedcba09-8765-4321-1234-567890abcdef --keyfile
```

```
~/examplekeys/benchmark_ecdsa_key.pub
```

8. Enable SSH tunneling on the container:

```
sudo /opt/nfast/bin/csadmin sshd state enable --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
SSHD PORT: 6789
LISTENING ADDRESS: aaaa::aa:aaaa:aaaa:aaaa
```

The output of `sshd state enable` contains the SSHD port number and the listening address of the container SSHD.

9. Start the container using `csadmin start`:

```
sudo /opt/nfast/bin/csadmin start --uuid fedcba09-8765-4321-1234-567890abcdef
```

Example output:

```
FEDC-BA09-8765        SUCCESS
IP ADDRESS: ffff::fff:ffff:ffff:ffff
```

The IPv6 address of the started container appears in the output of the `csadmin start` command. It can also be found in the output of `csadmin list` and `csadmin stats`.

10. Setup the SSH tunnel on the host:

Run `csadmin sshd state get` and collect the following information:

- Container tunnel address (`ffff::fff:ffff:ffff:ffff`)
- Container port (`8888`)
- SSHD port (`6789`)
- SSHD listening address (`aaaa::aa:aaaa:aaaa:aaaa`)

On nShield Connect the `SSHD listening address` may be an `IPv4` or `IPv6` address

Next, choose a local IP address and port number through which to access the tunnel. Typically localhost is chosen as the local IP address (`127.0.0.1` or `[::1]`)

The SSH tunnel command is formatted as follows:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L LOCAL_IP:LOCAL_PORT:[TUNNEL_ADDRESS%lxcbr0]:CONTAINER_PORT
-f -N -p SSHD_PORT launcher@LISTENING_ADDRESS
```

Using the example data:

```
ssh -i ~/examplekeys/helloworld_tcp_ecdsa_key -L [::1]:8888:[ffff::fff:ffff:ffff:ffff%lxcbr0]:8888 -f -N -p
6789 launcher@aaaa::aa:aaaa:aaaa:aaaa%nshield0
```

When using nShield 5s the IPv6 address is link-local and requires the zone index to be appended (typically `%nshield0`).

11. Run the host-side application.

    The host-side application takes three required optional arguments and two positional arguments. The required optional arguments are the IPv6 address set up in the forwarding above `[::1]` (`--ipv6`), the UUID of the container (`--uuid`), and the path to the signed `.cs5` image (`--cs5`). The required positional arguments are the simple signing key created earlier.

    ```
    ~/buildhost/n5/csee/benchmark/hostside/bm-test --uuid fedcba09-8765-4321-1234-567890abcdef --ipv6 ::1 --cs5
    ~/buildmodule/n5/csee/benchmark/module/bm-machine-signed.cs5 simple signingkeybenchmark
    ```

    Example output:

    ```
    Worldid: 0x1234abcd
    1 759 759.00
    2 1522 761.00
    3 2361 787.00
    4 3324 831.00
    5 4238 847.60
    6 5124 854.00
    7 5948 849.71
    8 6723 840.38
    9 7579 842.11
    10 8408 840.80
    ```

9. Build and sign example SEE machines on Windows

9.1. Prerequisites

- Visual Studio 2022 buildtools
- CMAKE version 3.9 or newer
- Ninja build system latest version
- Visual Studio 2022 workload-vctools

9.2. Building Windows CodeSafe C, CSEE, and NETSEE examples

1. Start the Developer Command Prompt for VS 2022 as Administrator from the **Start** menu.
2. Navigate to the following directory:

```
cd "c:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\Common7\Tools"
```

3. Install the MSVC C and C++ compiler `cl.exe`.
4. Execute `VsDevCmd.bat`:

```
VsDevCmd.bat
```

5. Run cl:

```
cl
```

6. Because the default is 32bit mode, the version displayed will show x86. Change to 64bit `cl` Compiler:

```
cd "c:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\VC\Auxiliary\Build"
```

7. Execute `vcvars64.bat`:

```
vcvars64.bat
```

8. Run `cl` and verify that the x64 version is displayed:

```
cl
```

you can build the following examples in the same VS2022 Command window:

9.2.1. Host-side examples

```
c:\>mkdir examples\host

c:\>cd c:\examples\host\

c:\examples\host>cmake -G Ninja -DCMAKE_C_COMPILER=cl -DCMAKE_CXX_COMPILER=cl "c:\Program
Files\nCipher\nfast\c\csd5\examples"

c:\examples\host>ninja
```

9.2.2. Module-side examples

```
c:\>mkdir examples\module

c:\>cd c:\examples\module\

c:\examples\module>cmake -G "Ninja" -DCMAKE_TOOLCHAIN_FILE="c:\Program Files\nCipher\nfast\c\csd5\cmake\codesafe-
toolchain-nshield5-csee.cmake" "c:\Program Files\nCipher\nfast\c\csd5\examples"

c:\examples\module>ninja
```

9.3. CS5 images for Python examples

Build the following images in the VS2022 Command window configured in Building Windows CodeSafe C, CSEE, and NETSEE examples. You do not need to build host-side and module-side Python examples separately. They are both built into `examples\python\n5\netsee\<example>\`.

```
c:\>mkdir examples\python

c:\>cd c:\examples\python\

c:\examples\python>cmake -G "Ninja" "c:\Program Files\nCipher\nfast\python3\csd5\examples"

c:\examples\python>ninja
```

For example:

```
c:\examples\python\n5\netsee\tickets>dir
Volume in drive C is OS
```

```
Volume Serial Number is 582A-CFB6 Directory of c:\examples\python\n5\netsee\tickets 03/21/2023  12:32 PM    <DIR>
.
03/21/2023  12:32 PM    <DIR>          ..
03/21/2023  12:32 PM    <DIR>          hostside
03/21/2023  12:32 PM    <DIR>          module
               0 File(s)              0 bytes
               4 Dir(s)  906,165,829,632 bytes free
```

9.4. Sign CodeSafe images

Signing CodeSafe Images requires a Security World and Operator Card Set (OCS).

1. Insert the OCS card.
2. Create a certificate signing request (CSR) that should be sent to Entrust to be signed:

```
c:\ca_ids\>csadmin ids create --keyname testdeveloperkey --x509cname developer.entrust.com --x509country US
--x509province FL --x509locality Shakopee --x509org Entrust --x509orgunit "Entrust CodeSafe"
Generate key 'testdeveloperkey' ...

Loading `TestOCS':
 Module 1: 0 cards of 1 read
 Module 1 slot 0: empty
Card reading complete.

OK
Generate a CSR in 'testdeveloperkey.csr' ...
OK
Created CSR file 'testdeveloperkey.csr'. Please send it to Entrust Support
```

The developer ID creation in this example was done with `TestOCS`, quorum of 1/1. Exact output may vary slightly with different OCS quorums.

3. Send the resulting CSR to customer support to be signed by Entrust. You must obtain the signed developer ID certificate in order to sign and load an application.

   For more detailed information on Developer IDs and CSRs, see Sign and deploy CodeSafe 5 SDK apps using csadmin.

4. Create the ASK on the HSM (the name of the key in this example is `test-ask`). The following example specifies the key to be protected by the module. However, end users are encouraged to protect the key with an OCS:

```
c:\ca_ids>C:\Progra~1\nCipher\nfast\bin\generatekey.exe --module=1 simple type=ECDSA curve=NISTP521
ident=test-ask plainname=test-ask
protect: Protected by? (token, module) [token] > module
nvram: Blob in NVRAM (needs ACS)? (yes/no) [no] >
key generation parameters:
 operation    Operation to perform       generate
```

```
 application  Application               simple
 protect      Protected by              module
 verify       Verify security of key    yes
 type         Key type                  ECDSA
 ident        Key identifier            test-ask
 plainname    Key name                  test-ask
 nvram        Blob in NVRAM (needs ACS) no
 curve        Elliptic curve            NISTP521
Key successfully generated.
Path to key: C:\ProgramData\nCipher\Key Management Data\local\key_simple_test-ask
```

5. Confirm that the keys were created in the previous step:

```
c:\ca_ids>nfkminfo -k
Key list - 2 keys
 AppName simple               Ident test-ask
 AppName simple               Ident testdeveloperkey
```

6. Sign the `netsee\tickets` example. You need the signed `cert.pem` from customer support for this step and the OCS card must be inserted for signing.

```
c:\examples\module\n5\netsee\tickets_netsee\module>csadmin image sign --askeyname test-ask --devkeyname
testdeveloperkey --devcert c:\ca_ids\testdeveloperid_cert.pem --out seetickets_netsee-signed-with-hsm.cs5
seetickets_netsee.cs5
INFO: Reading CS5 file contents...
INFO: Getting key handle from HSM...

INFO: Signing the Application Signing Key...
INFO: hashing contents using 'SHA512Hash'
INFO: Obtaining public key data from HSM...
INFO: Storing public key data on CS5 file...
INFO: Getting key handle from HSM...
INFO: Requesting signature from HSM...
INFO: Saving CS5 file to disk...
INFO: file 'seetickets_netsee.cs5' was signed successfully!

Directory of c:\examples\module\n5\netsee\tickets_netsee\module

02/16/2023  03:53 PM        27,167,860 seetickets_netsee-signed-with-hsm.cs5
               1 File(s)     27,167,860 bytes
               0 Dir(s)  775,613,321,216 bytes free
```

7. Install the developer ID certificate chain from Entrust using `csadmin ids add`:

```
csadmin ids add entrust_developerid_cert_chain.pem
FEDC-BA09-8765        SUCCESS

csadmin ids list
FEDC-BA09-8765        SUCCESS
Certificates:
{'serialNumber': '1', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust
CodeSafe, Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'abcdef12345678900987654321fedcbaabcdef12', 'authKeyid': '0987654321fedcbaabcdef123456789009876543',
'notBefore': '2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
{'serialNumber': '2', 'subject': 'Common Name: developer.entrust.com, Organizational Unit: Entrust
CodeSafe, Organization: Entrust, Locality: Shakopee, State/Province: Minnesota, Country: US', 'keyid':
'1234567890abcdeffedcba098765432112345678', 'authKeyid': 'fedcba0987654321123456789abcdeffedca098',
'notBefore': '2023-01-01 12:34:56+00:00', 'notAfter': '2024-01-01 12:34:56+00:00'}
```

8. Execute `netsee\tickets`:

```
c:\examples\module\n5\netsee\tickets_netsee\module>csadmin load seetickets_netsee-signed-with-hsm.cs5
FEDC-BA09-8765: Uploading seetickets_netsee-signed-with-hsm.cs5
FEDC-BA09-8765: creating machine
FEDC-BA09-8765        SUCCESS
UUID: fedcba09-8765-4321-1234-567890abcdef

c:\examples\module\n5\netsee\tickets_netsee\module>cd c:\examples\host\n5\netsee\tickets_netsee\hostside

c:\examples\host\n5\netsee\tickets_netsee\hostside>nopclearfail -aO
Module 1, command ClearUnitEx: OK

c:\examples\host\n5\netsee\tickets_netsee\hostside>csadmin start -u fedcba09-8765-4321-1234-567890abcdef
FEDC-BA09-8765        SUCCESS
IP ADDRESS: ffff::fff:ffff:ffff:ffff

c:\examples\host\n5\netsee\tickets_netsee\hostside>csadmin list
FEDC-BA09-8765
UUID                                  State     Name                IP Address
-------------------------------------------------------------------------------------------
fedcba09-8765-4321-1234-567890abcdef    RUNNING   seetickets_netsee   ffff::fff:ffff:ffff:ffff


c:\examples\host\n5\netsee\tickets_netsee\hostside>hosttickets_netsee.exe -p 8888 -U fedcba09-8765-4321-
1234-567890abcdef -i ffff::fff:ffff:ffff:ffff%10 -c
c:\examples\module\n5\netsee\tickets_netsee\module\seetickets_netsee-signed-with-hsm.cs5
WSAStartup() Success.
HostSide>Enter string to be encrypted (8 characters maximum): hello
HostSide>Reading Identities from container
HostSide>Generating RSA keypair
HostSide>Creating World: init status was 0 (OK)
HostSide>Sending ticket for private RSA key to module
HostSide>Sending key blob to module
HostSide>Sending cipher-text to module
HostSide>decrypted cipher text received from SEE machine:
"hello"
HostSide>Thank you for watching. The end.
```

10. Debug CodeSafe 5 SEE machines

`csadmin` exposes several commands you can use to manage SEE application logging.

The following SEE logging-related commands are supported by the `csadmin` utility.

10.1. config log set enabled

The `config log set enabled` command should be issued before the `start` command. It uses the following format:

```
/opt/nfast/bin/csadmin config set log enabled -u <SEE-machine-UUID> --esn <host-ESN>
```

- `<SEE-machine-UUID>` is the UUID of the SEE machine created by the load command.
- `<host-ESN>` is the ESN of the HSM hosting the SEE Machine.

For example:

```
/opt/nfast/bin/csadmin config set log enabled -u fedcba09-8765-4321-1234-567890abcdef --esn FEDC-BA09-8765
```

When successful, the command returns with no error.

10.2. config log set disabled

The `config log set disabled` command should be issued while the SEE machine is not running. It uses the following format:

```
/opt/nfast/bin/csadmin config set log disabled -u <SEE-machine-UUID> --esn <host-ESN>
```

- `<SEE-machine-UUID>` is the UUID of the SEE machine created by the load command.
- `<host-ESN>` is the ESN of the HSM hosting the SEE Machine.

For example:

```
/opt/nfast/bin/csadmin config set log disabled -u fedcba09-8765-4321-1234-567890abcdef --esn FEDC-BA09-8765
```

When successful, the command returns with no error.

10.3. log get

The `get` command returns the current SEE log contents, if any. It uses the following format:

```
/opt/nfast/bin/csadmin log get -u <SEE-machine-UUID>
```

`<SEE-machine-UUID>` is the UUID of the SEE machine created by the load command.

For example:

```
/opt/nfast/bin/csadmin log get -u fedcba09-8765-4321-1234-567890abcdef
FEDC-BA09-8765        SUCCESS
Success: Started ipcdaemon
```

10.4. log clear

The `clear` command deletes the current SEE log file if present. It uses the following format:

```
/opt/nfast/bin/csadmin log clear -u <SEE-machine-UUID>
```

`<SEE-machine-UUID>` is the UUID of the SEE machine created by the load command.

For example:

```
/opt/nfast/bin/csadmin log clear -u fedcba09-8765-4321-1234-567890abcdef
FEDC-BA09-8765        SUCCESS
log: log cleared
```

11. Uninstall the CodeSafe 5 SDK

Do not uninstall Security World or CodeSafe 5 software unless you are certain it is no longer required or you are going to upgrade it.
If you are using CodeSafe 5 with an nShield 5s HSM, you must back up its `sshadmin` keys by running `hsmadmin keys backup` before you uninstall Security World or CodeSafe 5.

The uninstaller only removes files that were created during the installation. To remove key data or Security World data, navigate to the installation directory and delete the files in the *%NFAST_KMDATA%* folder.

If you intend to remove your Security World before uninstalling the Security World Software, Entrust recommends that you erase the OCS before you erase the Security World or uninstall the Security World Software. Except where Remote Administration cards are used, after you have erased a Security World, you can no longer erase any cards that belonged to it.

1. Log in to the host computer as Administrator or as a user with local administrator rights.
2. Run the following command to erase the OCS:

   ```
   createocs -m# -s0 --erase
   ```

   Where # is the module number.

3. Uninstall the Security World and CodeSafe software:

   - **Linux**:

     Run the following command:

     ```
     /opt/nfast/sbin/install -u
     ```

   - **Windows**:

     1. Navigate to the Windows Control Panel, and select **Programs and Features**.
     2. Select the Security World Software entry, then select **Uninstall** to remove the software.

If required, you can safely remove the nShield module after shutting down all connected hardware.

12. Port existing CodeSafe application to CodeSafe 5

Follow the steps in this chapter if you need to port an existing legacy SEE machine to run on CodeSafe 5.

The porting of legacy CodeSafe application examples in this chapter assumes the perspective of a CodeSafe application developer. CodeSafe users wanting to port legacy third party CodeSafe applications to nShield 5 might need to have the third party issuer of said legacy CodeSafe applications port the applications and sign the ported applications.

CodeSafe users porting third party applications should ensure that the third party CodeSafe developer is a trusted party, and should verify that the ported CodeSafe image has a genuine certificate issued by the trusted developer. After a third party CodeSafe application is ported and signed, the application user can skip to the "Load the signed container" step in the following examples and continue the procedures from there.

Full examples of legacy SEE machines that have been ported with use of the compatibility layer can be found in Build and sign example SEE machines on Linux. These Classic SEE "CSEE" examples are legacy examples that have been modified to run with CodeSafe 5 specifically to demonstrate use of the compatibility layer. In all other ways, these examples are identical to examples provided with previous iterations of nShield HSMs and CodeSafe.

It is assumed that an ASK and developer ID key have already been generated, and that required certificates have already been obtained from Entrust and installed into the target HSM.

12.1. The compatibility layer

Legacy CodeSafe transacted data between host application and module SEE machines using SEEJobs. SEEJobs were sent from the host-side application to the nCore API which then passed the jobs on to the SEE machine, and vice versa. CodeSafe 5 removes the need to communicate with SEE machines via the nCore API using SEEJobs.

Instead, CodeSafe 5 allows a network connection to be established directly between a host-side application and an SEE machine. As such, support for transacting SEEJobs, and all related methods has been removed from CodeSafe 5.

The compatibility layer provides support for SEEJobs. All methods that dealt with transacting exist in the compatibility layer, but instead of passing SEEJobs to the nCore API and having the nCore API forward them, the Compatibility layer creates a network

connection between the SEE machine and host application.

The methods function similarly, but the mechanism for data transaction has been updated. The compatibility layer is split into two parts: the module-side compatibility layer, and the host-side compatibility layer. Both parts work together to provide support for legacy SEE machines.

The module-side SEE machine and corresponding host-side application must both be ported successfully for them to function on CodeSafe 5. It is not sufficient to port one side but not the other.

12.1.1. Module-side compatibility layer

The module-side compatibility layer provides the methods necessary to connect the SEE machine to the host-side application via network connection.

The module-side compatibility layer comprises the `liblegacy_compatibility.a` library. Its install location is:

- **Linux**: `/opt/nfast/c/csd5/lib-ppc64-linux-musl/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\lib-ppc64-linux-musl\`

12.1.2. Host-side compatibility layer

The host-side compatibility layer provides the methods necessary to connect the host-side application to the SEE machine via network connection.

The host-side compatibility layer comprises the following files:

- `legacy-csee-host-side-compatibility.h`
- `legacy-csee-host-side-compatibility.c`

Their install location is:

- **Linux**: `/opt/nfast/c/csd5/examples/csee/utils/hostside/`
- **Windows**: `C:\Program Files\nCipher\nfast\c\csd5\examples\csee\utils\hostside\`

12.2. Required module-side changes for porting

To port a legacy SEE machine to CodeSafe 5, only a single line change is required in code.

Initialize the compatibility layer by calling `SEElib_Legacy_Support_Init()` *after*

`SEElib_Init()` is called but *before* any legacy methods such as `SEElib_AwaitJob()` are called. This waits for a compatible host-side application to connect before proceeding.

An example SEE machine `main()` properly initializing the compatibility layer:

```
int main(void) {

  /* initialize the SEE environment */
  SEElib_init();

  /* initialize legacy SEE support */
  SEElib_Legacy_Support_Init("8888");

  /* The compatibility layer is initialized
     carry on with SEE machine operation */
  Perform_SEE_Machine_Tasks();
  return 0;
}
```

By default, all provided example SEE machines communicate through port `8888`. You can use any port when initializing the compatibility layer, however you must ensure that the host-side application compatibility layer is passed and attempts to connect to the same port number as the one initialized on the module-side.

After the compatibility layer has been initialized, all SEEJob-related methods, such as `SEElib_ReturnJob()` or `SEElib_AwaitJob()`, will work. No further changes in code are required for legacy SEE machines to run using CodeSafe 5.

A full list of methods the compatibility layer provides support for can be found in the "SEE Machine Module Side Compatibility Layer" section of SEE API documentation.

12.3. Required host-side changes for porting

Porting host-side applications to CodeSafe 5 requires changes to some method calls, in addition to the initialization.

12.3.1. Initialization

Initialize the host-side compatibility layer using the following command:

```
netsee_initialize_legacy_seejob_support(const char * cseeContainerMachineIPv6, const char *
cseeContainerMachinePort)`
```

It takes two arguments:

- The SEE machine container IP, which can be found using `csadmin list`
- The SEE machine port number

The port number must match the port number passed on to the module-side compatibility layer when the module-side compatibility layer is initialized.

The container IP must be passed to the host-side application. You can pass it in as a command line argument, as in the classic SEE examples described in Build and sign example SEE machines on Linux, however the exact implementation is the decision of the porting developer.

12.3.2. Replacing SEEJob-related method calls

Unlike the module-side compatibility layer, which allows all SEEJob-related method calls to be called without changes, porting the host-side requires certain method calls to be updated.

This is because the compatibility layer's replacement methods need to replace the role of the nCore API and send SEEJobs to and from the SEE machine's module-side compatibility layer.

The methods specific to the nCore API that host-side applications previously used to transact SEEJobs still exist to communicate with the nCore API, but no longer support SEEJobs.

Only nCore API calls for SEEJobs need to be updated. Other unrelated calls to the nCore API do not need to be modified.

Host-side compatibility calls no longer require the `NFastApp_Connection` and `NFast_Call_Context` arguments to be passed in, as demonstrated in the following examples.

For more detailed descriptions of these methods, see the "Compatibility layer API Host-side" section of SEE API documentation.

12.3.2.1. NFastApp_Submit()

Replace SEEJob calls to `NFastApp_Submit()` with calls to `netsee_submit_legacy_seejob()`.

For example:

```
NFastApp_Submit(nc, NULL, &cmd, &reply, &tctx);
```

Becomes:

```
netsee_submit_legacy_seejob(&cmd, &reply, &tctx);
```

12.3.2.2. NFastApp_Wait()

Replace SEEJob calls to `NFastApp_Wait()` with calls to `netsee_wait_legacy_seejob()`.

For example:

```
NFastApp_Wait(conn, NULL, &replyp, &tctxp);
```

Becomes:

```
netsee_wait_legacy_seejob(&replyp, &tctxp);
```

12.3.2.3. NFastApp_Transact()

Replace SEEJob calls to `NFastApp_Transact()` with calls to `netsee_transact_legacy_seejob()`.

For example:

```
NFastApp_Transact(conn, NULL, &cmd, &reply, &tctx);
```

Becomes:

```
netsee_transact_legacy_seejob(&cmd, &reply, &tctx);
```

12.3.2.4. simple_transact()

Replace SEEJob calls to `simple_transact()` with calls to `netsee_simple_transact_legacy_seejob()`.

For example:

```
simple_transact(conn, NULL, &cmd, &reply, 1);
```

Becomes:

```
netsee_simple_transact_legacy_seejob(&cmd, &reply, 1);
```

12.4. Rebuilding and Recompiling

After the host-side application and module-side SEE machine compatibility layers have been properly initialized, and all host-side SEEJob method replacements have been made in the code, both the host-side application and the module-side SEE machine should be rebuilt with their respective compatibility layers properly linked and included.

The provided Classic SEE examples are practical examples of how the compatibility layer should be implemented, and how the compatibility layer libraries and files should be linked the build chain creating the SEE machine.

12.4.1. Rebuilding host-side

Include `legacy-csee-host-side-compatibility.h` in host-side application scripts that are being ported. Recompile host-side applications so that `legacy-csee-host-side-compatibility.c` is included in the source.

12.4.2. Rebuilding Module Side

Link the compatibility layer library `liblegacy_compatibility.a` to the module-side SEE machine after the changes to the SEE machine source code have been made to initialize the compatibility layer.

13. Supporting legacy CodeSafe Direct

CodeSafe Direct is no longer available in CodeSafe 5. The following sections describe the usage of legacy CodeSafe Direct and how similar functionality is accomplished via CodeSafe 5.

13.1. Legacy CodeSafe Direct

Originally, the application would connect to the HSM through the Security World hardserver. With legacy CodeSafe Direct, the nShield Connect could be configured to receive direct socket connections to the SEE machine via see-sock-serv, removing the need for a client machine. You could 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.

13.2. CodeSafe 5

The CodeSafe 5 modern architectural approach provides a container which has an IPC daemon (UNIX domain socket) that is used to send and receive nCore API commands and replies. The communication between the host application and CodeSafe 5 container is provided by a secure SSH daemon making use of port forwarding.

The `Cmd_SEEJob` nCore API command is no longer supported by the nCoreAPI service. Instead, the command is now requested directly from the client application on the host to the SEE machine using a direct TCP connection. A support library is needed to support this new connection, and this is part of the compatibility layer.

Containers listening on a specific port via the secure channel is a 'CodeSafe Direct' replacement.

There are cli commands using the 'csadmin' utility that can establish the secure SSHD port forwarding on the host client machine. The `cs5-port-monitor` will validate and then forward the ports specified in `network-conf.json`. See Build and sign example SEE machines on Linux for examples of using an SSH tunnel to communicate between the client and SEE machine directly through a TCP/IPv6 network connection to the container. Containers can be configured to listen to ports using the `network-conf.json` file.

14. SEE API documentation

`SEElib` is an API that enables an SEE machine to execute nCore API commands. Historically, the `SEElib` also provided the functionality which connected SEE machines to their host-side applications via the nCore API. In CodeSafe 5, `SEElib` still provides the methods necessary to execute nCore API commands, but communication between the SEE machine and the host-side application is expected to be done using TCP/IPv6 network connections which are managed directly by the SEE machine. To allow for a more seamless integration of legacy SEE machines, which previously transacted with their host-side application via the nCore API, a compatibility layer has been created to automatically manage these legacy transactions.

The `SEElib` API is provided as a library `seelib.a` that can be found in the rootfs after install. Its install location is `/opt/nfast/c/csd5/lib-ppc64-linux-musl/seelib.a` on Linux.

14.1. Why CodeSafe 5 needs a compatibility layer

The compatibility layer allows pre-existing CodeSafe users to port legacy SEE machines that were developed for nShield XC or Solo+ HSMs to the CodeSafe 5 environment.

CodeSafe 5 has a Launcher service for managing the SEE container, instead of using nCore API commands. All requests related to container (SEE machine) management, for example to load a new SEE machine onto the HSM or to start, stop, or destroy a SEE machine, are made directly to the new Launcher service.

Legacy `SEElib` applications previously allocated memory by the `Cmd_CreateSEEWorld` nCore API command. In CodeSafe 5, `launcher receive`, `launcher create`, and `launcher start` requests are made to the Launcher service in combination with a new `Cmd_CreateSeeConnection` command to the nCore API service to get a SEE machine running and able to communicate with the nCore API service.

For CodeSafe 5 applications, the nCore API service does not support the `Cmd_SEEJob` nCore API command. Instead, the command is requested directly from the client application on the host to the SEE machine using a direct TCP/IPv6 network connection. The compatibility layer provides support for this new connection method.

CodeSafe 5 does not use the concept of `UserData`. A developer can include any files, using any directory structure, in the container image that is installed in the HSM.

14.2. SEElib functions

14.2.1. SEElib_init

```
extern void SEElib_init(void);
```

This function initializes the SEElib library.

This function does not return on error.

14.2.2. 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.

UserData in CodeSafe 5 is a file located inside the container (/etc/codesafe.userdata) and must be added when the image is constructed.

14.2.3. SEElib_ReleaseUserData

```
extern void SEElib_ReleaseUserData(void);
```

In CodeSafe 5 this function does not do anything. It is only present to satisfy the linker.

14.2.4. SEElib_InitComplete

```
extern void SEElib_InitComplete( M_Word status );
```

In CodeSafe 5 this function does not do anything. It is only present to satisfy the linker.

14.2.5. 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.

14.2.6. 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.

14.2.7. 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 CodeSafe 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.

14.2.8. 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.

14.2.9. 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*).

14.2.10. 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*).

14.2.11. 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).

14.2.12. 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).

14.2.13. SEElib_GetUserDataLen

```
extern M_Word SEElib_GetUserDataLen ( void );
```

In CodeSafe 5, this function gets the length in bytes of the `/etc/userdata.codesafe` file in the filesystem of the container.

If this data has been discarded because `SEElib_ReleaseUserData()` has been called, this function returns `0`.

14.2.14. 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 unmarshalled 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.

14.2.15. 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 listener 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` can 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, `TransactionNotYetComplete` 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.

14.3. About the SEElib compatibility layer

The compatibility layer is provided to help port existing SEE machines and their host-side applications to the new CodeSafe 5 architecture. The compatibility layer provides support for legacy methods that dealt with the host-side application/SEE machine connection (sending SEEJobs between the two and their supporting methods). Because the new CodeSafe 5 architecture has removed the need to send SEEJobs between the host-side application and the SEE machine by using the nCore API as an intermediary, these methods are no longer found in the CodeSafe 5 `SEElib` API.

For detailed examples of the SEElib compatibility layer's use, refer to the provided "CSEE" or "Classic SEE" examples. These examples are legacy SEE machine examples that have been ported using the compatibility layer.

14.4. SEE machine module side compatibility layer

The module-side compatibility layer provides a small API to emulate the deprecated CSEE methods while using the CodeSafe 5 architecture and TCP/IPv6 network connections underneath.

To continue to use legacy methods within an SEE machine, the SEE machine must be recompiled with the compatibility layer library: `liblegacy_compatibility.a`. The default install location is `/opt/nfast/c/csd5/lib-ppc64-linux-musl/liblegacy_compatibility.a` on Linux. This library provides support for the legacy SEElib methods described below.

There is only a one-line change that needs to be made within an SEE machine's source to initialize the compatibility layer. A call to `SEElib_Legacy_Support_Init()`. This call must be made before any of the legacy `SEElib` calls are made, typically in `main()` after `SEElib_init()`. After this call is made, all legacy methods operate functionally identically to legacy versions of CodeSafe, while using TCP/IPv6 network connections behind the scenes.

Do not write new applications using the compatibility layer. The compatibility layer is provided to simplify the porting of existing legacy applications to CodeSafe 5.

CodeSafe 5 allows the use of TCP/IPv6 network connections to connect the host-side application to an SEE machine, simplifying the communication between the two, and expanding the functionality of the communication between the two. The compatibility layer allows legacy applications to run using the old style of SEEJobs, but doing so with new applications is not advised.

14.4.1. SEElib_Legacy_Support_Init

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern void SEElib_Legacy_Support_Init(const char* PORT);
```

This function initializes the compatibility layer for legacy SEE machines for use with CodeSafe 5. This method must be called before any other legacy methods. This method initializes all the support required for legacy SEE machines to function properly.

14.4.2. SEElib_AwaitJob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int SEElib_AwaitJob( M_Word *tag_out, unsigned char *buf, M_Word *len_io );
```

This function blocks and waits for the next `SEEJob` to come in from the host-side application. 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 of the `SEEJob` that was sent by the host-side application.

The `*tag_out` value is the tag for this command. Each transaction must have a unique tag when sent from the host-side application to ensure transactions are returned to their required caller. The generation of unique tags is handled by the host-side compatibility layer. The tag must be returned in the `SEElib_ReturnJob` so that the host-side compatibility layer associates the reply with this transaction.

The `SEElib_AwaitJob` function returns an `M_Status`, which is `OK` on success and normally, but not always, `BufferFull` on failure.

If you use `SEElib_StartProcessorThreads()`, these function calls are done automatically and you should not call this function yourself.

14.4.3. SEElib_AwaitJobEx

```
extern void SEElib_AwaitJobEx( M_Word *tag_out, unsigned char *buf, M_Word *len_io, unsigned flags );
```

Block on the socket waiting for a SEEJob command from the host.

The output parameters are filled with information obtained from the message itself. 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 of the SEEJob command. The *tag_out value is the tag for this command.

14.4.4. SEElib_ReturnJob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern void SEElib_ReturnJob( M_Word tag, const unsigned char *data, unsigned int len );
```

This function returns an SEEJob reply to the host-side application. It is sent in a way that the host-side compatibility layer can interpret and write into the corresponding reply struct on the host-side.

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 on the host-side application.

14.4.5. SEElib_StartProcessorThreads

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
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 compatibility layer 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 a 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 for information only. The function processes 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.

14.4.6. SEElib_StartSEEJobListener

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
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_ReleaseSEEJob` to free the buffer.

It is safe to call this function multiple times, however calls after the first call have no effect.

14.4.7. SEElib_QuerySEEJob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
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`.

14.4.8. SEElib_ReleaseSEEJob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern void SEElib_ReleaseSEEJob( unsigned char **buf );
```

This function is called to release a buffer which was returned from `SEElib_QuerySEEJob`. It 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, it sets `*buf` to NULL on completion.

14.5. Compatibility layer API Host side

Legacy host-side applications need to be modified to use the network interface to talk the SEE machine instead of the nCore API. The bulk of this work is handled automatically by including the host-side compatibility layer and recompiling. However, all calls to the nCore API which use `CMD_SEEJob` need to be modified slightly to reference the new CodeSafe 5 compatible methods. The compatibility layer provides support to emulate the use cases of

the Cmd_SEEJob message interface. The compatible calls and the methods they replace are described below. All other calls by the host-side application to the nCore API will remain unchanged.

14.5.1. netsee_initialize_legacy_seejob_support

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int netsee_initialize_legacy_seejob_support(const char * cseeContainerMachineIPv6, const char *
cseeContainerMachinePort);
```

This function initializes host-side application compatibility layer to support legacy CodeSafe SEEJob commands. netsee_initialize_legacy_seejob_support() must be called to initialize legacy support for CodeSafe 5. The call creates all necessary processor threads, initializes all values and fields required to process SEEJob M_Commands, and creates a connection to the SEE machine via TCP/IPv6 networking. This call must be made before any of the other methods described below are called.

14.5.2. netsee_submit_legacy_seejob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int netsee_submit_legacy_seejob(const M_Command *cmd, M_Reply *reply, struct NFast_Transaction_Context
*tctx);
```

This function transmits a SEEJob command to the SEE application.

Replaces NFastApp_Submit().

The compatibility layer strips the relevant SEEJob information from the M_Command, issues a unique tag, and marshals this information to a form the compatibility layer compiled SEE machine understands. It then sends the command to the module directly via a TCP/IPv6 connection initialized by the compatibility layer.

14.5.3. netsee_wait_legacy_seejob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int netsee_wait_legacy_seejob(M_Reply **replyp, struct NFast_Transaction_Context **tctx);
```

This function waits to receive a reply from the SEE machine.

Replaces `NFastApp_Wait()`.

The compatibility layer reads an incoming reply from the module, parses the information, and writes it to the correct `M_Reply` corresponding to the tag the command was sent with. It does not proceed beyond the call until this reply has been processed. After a reply is received and marshaled by the compatibility layer, `netsee_wait_legacy_seejob()` will return with the correct reply.

14.5.4. netsee_transact_legacy_seejob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int netsee_transact_legacy_seejob(const M_Command *command, M_Reply *reply, struct
NFast_Transaction_Context *tctx);
```

This function transacts a `SEEJob` command and waits until a reply is received and written to `*reply`.

Replaces `NFastApp_Transact()`.

The compatibility layer strips the relevant `SEEJob` information from the `M_Command`, issues a unique tag, and marshals this information to a form the compatibility layer compiled SEE machine understands. It then sends the command to the module SEE machine directly via a TCP/IPv6 connection initialized by the compatibility layer.

After sending the command, it waits for a reply from the SEE machine via the established network connection. The compatibility layer reads the incoming reply from the module, parses the information, and writes it to the correct `M_Reply` corresponding to the tag the command was sent with.

After a reply is received and marshaled by the compatibility layer, `netsee_transact_legacy_seejob()` returns with the correct `M_Reply` having been written to

`*reply`.

14.5.5. netsee_simple_transact_legacy_seejob

This function is provided by the compatibility layer to ease porting applications from Solo XC to nShield 5. Do not use it for new applications.

```
extern int netsee_simple_transact_legacy_seejob(const M_Command *cmd, M_Reply *reply, int fatal);
```

Transact a `SEEJob` command and wait until a reply is received and written to `*reply`. If `fatal` is true, and an error occurs, `exit(4)`.

Replaces `simple_transact()`.

The compatibility layer strips the relevant `SEEJob` information from the `M_Command`, issues a unique tag, and marshals this information to a form the compatibility layer compiled SEE machine will understand. It then sends the command to the module SEE machine directly via a TCP/IPv6 connection initialized by the compatibility layer. Then, it waits for a reply from the SEE machine via the established network connection. The compatibility layer reads the incoming reply from the module, parses the information, and writes it to the correct `M_Reply` corresponding to the tag the command was sent with. Once a reply is received and marshaled by the compatibility layer, `netsee_simple_transact_legacy_seejob()` will return with the correct `M_Reply` having been written to `*reply`.

15. System calls allowed by CodeSafe 5 SEE machines

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	20 __NR_getpid
21 __NR_mount	22 __NR_umount
24 __NR_getuid	29 __NR_pause
33 __NR_access	36 __NR_sync
37 __NR_kill	38 __NR_rename
39 __NR_mkdir	40 __NR_rmdir
41 __NR_dup	42 __NR_pipe
45 __NR_brk	47 __NR_getgid
49 __NR_geteuid	50 __NR_getegid
54 __NR_ioctl	55 __NR_fcntl
60 __NR_umask	63 __NR_dup2
64 __NR_getppid	65 __NR_getpgrp
66 __NR_setsid	78 __NR_gettimeofday
83 __NR_symlink	85 __NR_readlink
88 __NR_reboot	90 __NR_mmap
91 __NR_munmap	94 __NR_fchmod

Allowed system calls	
99 __NR_statfs	102 __NR_socketcall
106 __NR_stat	107 __NR_lstat
108 __NR_fstat	114 __NR_wait4
119 __NR_sigreturn	120 __NR_clone
122 __NR_uname	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	167 __NR_poll
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
185 __NR_sigaltstack	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
229 __NR_io_getevents	232 __NR_set_tid_address
234 __NR_exit_group	246 __NR_clock_gettime
250 __NR_tgkill	252 __NR_statfs64
281 __NR_ppoll	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

Allowed system calls	
337 __NR_recvfrom	338 __NR_shutdown
339 __NR_setsockopt	340 __NR_getsockopt
341 __NR_sendmsg	342 __NR_recvmsg
343 __NR_recvmmsg	344 __NR_accept4
349 __NR_sendmmsg	359 __NR_getrandom (See note)
365 __NR_membarrier	

`getrandom` is not implemented in nShield 5. Use either `/dev/random` or the `Cmd_GenerateRandom` nCore command instead.